

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/CI/NR 88-80	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
TITLE (and Subtitle) ASYNCHRONOUS DIGITAL REGULATORS		5. TYPE OF REPORT & PERIOD COVERED PHD THESIS
AUTHOR(s) VERNON SCOTT RITCHEY		6. PERFORMING ORG. REPORT NUMBER
PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: STANFORD UNIVERSITY		8. CONTRACT OR GRANT NUMBER(s)
CONTROLLING OFFICE NAME AND ADDRESS		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
		12. REPORT DATE 1988
		13. NUMBER OF PAGES 168
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) AFIT/NR Wright-Patterson AFB OH 45433-6583		15. SECURITY CLASS. (of this report) UNCLASSIFIED
16. DISTRIBUTION STATEMENT (of this Report) DISTRIBUTED UNLIMITED: APPROVED FOR PUBLIC RELEASE		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) SAME AS REPORT		
18. SUPPLEMENTARY NOTES Approved for Public Release: JAW AFR 190-8 LYNN E. WOLAVER <i>Lynn Wolaver</i> Dean for Research and Professional Development Air Force Institute of Technology Wright-Patterson AFB OH 45433-6583 19 July 88		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) ATTACHED		

DTIC  
ELECTE  
AUG 03 1988

# Asynchronous Digital Regulators

Vernon Scott Ritchey, Ph.D.

Stanford University, 1987

This report presents methods for designing and analyzing multirate asynchronous digital controls for linear systems. Multirate digital control is a natural approach for systems with widely-spaced natural frequencies. An asynchronous architecture provides a simple approach for assigning control tasks to distributed processors.

Previous multirate design methods required either synchronized samplers or high sample rates. Synchronized samplers produce a system that is periodically time varying. Alternatively, high sample rates simulate a continuous controller. In practice, synchronized implementations and implementations with high sample rates may have higher cost, complexity, and weight and lower reliability compared to asynchronous designs. In some cases, an asynchronous implementation with slow sampling will perform as well as a fast, synchronized design. The goal of this research was to develop methods to design and evaluate asynchronous control systems operating at minimal sample rates.

The multirate asynchronous design and analysis methods developed in this report use a time-domain approach based on the closed-loop state transition matrix. Design and analysis algorithms (implemented in PC-MATLAB) are included in an Appendix. The analysis is based on a sufficient stability criterion which gives an objective measure of long-term stability and indicates short-term stability. The design method allows the designer to specify the form of the controller. Numerical optimization is used to minimize a quadratic cost integral. Design and analysis examples are presented for a double integrator plant and a two-link robot arm.

Approved for Publication:

By

  
For Major Department

By

Dean of Graduate Studies

# ASYNCHRONOUS DIGITAL REGULATORS

A DISSERTATION  
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING  
AND THE COMMITTEE ON GRADUATE STUDIES  
OF STANFORD UNIVERSITY  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS  
FOR THE DEGREE OF  
DOCTOR OF PHILOSOPHY

By  
Vernon Scott Ritchey  
June 1987

Preceding Page/s BLANK In Document



Accession For	
NTIS CRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

**CERTIFICATE OF FINAL READING OF DISSERTATION**

Each doctoral dissertation, when submitted to the Graduate Program Office, must be accompanied by this page, with the signature of one of the readers who signed the dissertation.

It is hoped that this final reading will ensure that (1) footnotes and bibliography are in appropriate and consistent form; (2) all required illustrative tables and charts are in place; (3) all suggested changes have been incorporated in the manuscript; and (4) the dissertation is booklike in appearance and ready for binding and microfilming.

---

**CERTIFICATE OF FINAL READING OF DISSERTATION**

To the University Committee on Graduate Studies:

I certify that I have read the dissertation of

Vernon Scott Ritchey

in its *final* form for submission and have found it to be satisfactory.

Gene L. Franklin  
(SIGNATURE)

May 22, 1987  
(DATE)

# Asynchronous Digital Regulators

Vernon Scott Ritchey, Ph.D.

Stanford University, 1987

This report presents methods for designing and analyzing multirate asynchronous digital controls for linear systems. Multirate digital control is a natural approach for systems with widely-spaced natural frequencies. An asynchronous architecture provides a simple approach for assigning control tasks to distributed processors.

Previous multirate design methods required either synchronized samplers or high sample rates. Synchronized samplers produce a system that is periodically time varying. Alternatively, high sample rates simulate a continuous controller. In practice, synchronized implementations and implementations with high sample rates may have higher cost, complexity, and weight and lower reliability compared to asynchronous designs. In some cases, an asynchronous implementation with slow sampling will perform as well as a fast, synchronized design. The goal of this research was to develop methods to design and evaluate asynchronous control systems operating at minimal sample rates.

The multirate asynchronous design and analysis methods developed in this report use a time-domain approach based on the closed-loop state transition matrix. Design and analysis algorithms (implemented in PC-MATLAB) are included in an Appendix. The analysis is based on a sufficient stability criterion which gives an objective measure of long-term stability and indicates short-term stability. The design method allows the designer to specify the form of the controller. Numerical optimization is used to minimize a quadratic cost integral. Design and analysis examples are presented for a double integrator plant and a two-link robot arm.

Approved for Publication:

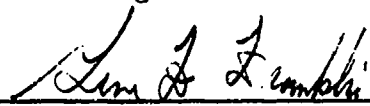
By

  
For Major Department

By

\_\_\_\_\_  
Dean of Graduate Studies

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

  
\_\_\_\_\_  
Gene F. Franklin  
(Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

  
\_\_\_\_\_  
Arthur E. Bryson

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

  
\_\_\_\_\_  
Peter M. Banks

Approved for the University Committee on Graduate  
Studies:

\_\_\_\_\_  
Dean of Graduate Studies

# Abstract

This report presents methods for designing and analyzing multirate asynchronous digital controls for linear systems. Multirate digital control is a natural approach for systems with widely-spaced natural frequencies. An asynchronous architecture provides a simple approach for assigning control tasks to distributed processors.

Previous multirate design methods required either synchronized samplers or high sample rates. Synchronized samplers produce a system that is periodically time varying. Alternatively, high sample rates simulate a continuous controller. In practice, synchronized implementations and implementations with high sample rates may have higher cost, complexity, and weight and lower reliability compared to asynchronous designs. In some cases, an asynchronous implementation with slow sampling will perform as well as a fast, synchronized design. The goal of this research was to develop methods to design and evaluate asynchronous control systems operating at minimal sample rates.

The multirate asynchronous design and analysis methods developed in this report use a time-domain approach based on the closed-loop state transition matrix. Design and analysis algorithms (implemented in PC-MATLAB) are included in an Appendix. The analysis is based on a sufficient stability criterion which gives an objective measure of long-term stability and indicates short-term stability. The design method allows the designer to specify the form of the controller. Numerical optimization is used to minimize a quadratic cost integral. Design and analysis examples are presented for a double integrator plant and a two-link robot arm.

# Acknowledgments

I sincerely wish to thank my advisor, Professor Gene F. Franklin, for his support, encouragement, and guidance throughout my research at Stanford.

I also thank the other members of my reading committee, Professors Arthur E. Bryson and Peter M. Banks, for their diligent review and constructive comments.

I especially thank my fellow students for their support and friendship. Their assistance and encouragement have been invaluable throughout my study at Stanford.

I thank the U.S. Air Force for the opportunity and financial support to pursue this program.

Finally, I thank my wife for her patience and understanding throughout these last few years.



# Contents

<b>Abstract</b>	<b>iii</b>
<b>Acknowledgments</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Purpose . . . . .	1
1.2 Related Literature . . . . .	2
1.3 Overview . . . . .	2
1.4 Scope of Methods . . . . .	4
1.5 Contributions . . . . .	4
<b>2 Problem Description</b>	<b>7</b>
2.1 Motivation . . . . .	7
2.2 Asynchronous Definitions . . . . .	8
2.2.1 Simultaneous Sample Processes . . . . .	8
2.2.2 Synchronous and Asynchronous Sampling . . . . .	9
2.2.3 Basic Time Period . . . . .	10
2.2.4 Phase . . . . .	10
2.2.5 BTP Segments and Events . . . . .	11
2.3 Sample Rate Considerations . . . . .	11
2.4 Limitations of Existing Design Methods . . . . .	12
2.4.1 Ad Hoc Approaches . . . . .	12
2.4.2 Exact Methods . . . . .	14
2.4.3 Common Limitations . . . . .	17

<b>3</b>	<b>Definitions, Conventions, and Notation</b>	<b>19</b>
3.1	Synchronous Sampling . . . . .	19
3.2	Basic Time Period . . . . .	19
3.2.1	Synchronous Basic Time Period . . . . .	19
3.2.2	Asynchronous Basic Time Period . . . . .	20
3.2.3	Sequence Phasing . . . . .	20
3.3	Kalman-Bertram Representation . . . . .	21
3.3.1	The State Vector . . . . .	21
3.3.2	State Transition Matrices . . . . .	22
3.3.3	BTP State Transition Matrix . . . . .	24
3.4	Asynchronous System Description . . . . .	24
3.5	Cost Function . . . . .	26
3.6	Process Noise . . . . .	28
3.7	Cost Function Scope . . . . .	28
<b>4</b>	<b>Asynchronous Stability Condition</b>	<b>31</b>
4.1	Stability Definition . . . . .	31
4.2	Factors of $\Psi$ . . . . .	32
4.3	General Sufficient Condition . . . . .	33
4.4	Asynchronous Multirate Stability . . . . .	34
4.4.1	An Important Special Case . . . . .	35
4.5	Eigenvector Scaling . . . . .	35
4.6	Analysis of Result . . . . .	35
<b>5</b>	<b>Stability Examples</b>	<b>39</b>
5.1	The System . . . . .	39
5.2	Nominal Sampling . . . . .	40
5.3	Asynchronous Sampling . . . . .	41
5.3.1	Analysis for $BTP=1$ . . . . .	41
5.3.2	Analysis for Synchronous BTP (9.0) . . . . .	41

<b>6</b>	<b>Synchronous Design Method</b>	<b>43</b>
6.1	Cost and Gradient . . . . .	43
6.1.1	Initial Cost Function . . . . .	44
6.1.2	Initial Gradient . . . . .	44
6.1.3	Main Cost Function . . . . .	45
6.1.4	Gradient of Main Cost Function . . . . .	51
6.2	Search Algorithm . . . . .	52
<b>7</b>	<b>Asynchronous Design Method</b>	<b>55</b>
7.1	Synchronous with Random Phase . . . . .	55
7.1.1	One Random Phase . . . . .	56
7.1.2	Design Algorithm . . . . .	57
7.2	Asynchronous Design . . . . .	58
<b>8</b>	<b>Design Examples</b>	<b>59</b>
8.1	Double Integrator Examples . . . . .	59
8.1.1	Nominal Case . . . . .	60
8.1.2	Synchronous Case . . . . .	61
8.1.3	Asynchronous Case . . . . .	62
8.1.4	Sample Rate Effects . . . . .	65
8.2	Two-Link Arm Examples . . . . .	70
8.2.1	Nominal Case . . . . .	72
8.2.2	Asynchronous Case . . . . .	73
<b>9</b>	<b>Practical Considerations</b>	<b>77</b>
9.1	BTP Selection . . . . .	77
9.1.1	Rule of Thumb . . . . .	78
9.1.2	Rationale . . . . .	78
9.2	Asynchronous Analysis of Synchronous Systems . . . . .	79
9.3	Accelerated Convergence . . . . .	80
9.4	Cost Weighting Matrices . . . . .	80
9.5	Numerical Properties . . . . .	81

9.6	PC-MATLAB Implementation . . . . .	82
9.6.1	Square Root Algorithms . . . . .	82
10	Summary and Recommendations . . . . .	85
10.1	Summary . . . . .	85
10.2	Recommendations for Future Research . . . . .	85
A	Eigensystem Derivatives . . . . .	87
A.1	Problem Statement . . . . .	87
A.2	Equation Development . . . . .	88
A.2.1	Eigenvector Derivatives . . . . .	89
A.3	The Singularity Condition . . . . .	90
B	Discrete Conversion . . . . .	93
B.1	Problem Statement . . . . .	93
B.2	Van Loan Results . . . . .	94
B.3	Main Result . . . . .	95
B.4	Proofs . . . . .	95
B.4.1	$\Phi$ Partial . . . . .	96
B.4.2	$Q$ and $R$ Partial . . . . .	97
C	STM Continuity . . . . .	99
C.1	The Phase Condition . . . . .	99
C.2	Boundary Conditions . . . . .	101
C.2.1	BTP Truncation Cases . . . . .	101
C.2.2	Commutivity Cases . . . . .	101
C.2.3	Recap . . . . .	102
D	Theorem Proofs . . . . .	103
D.1	Notation and Definitions . . . . .	103
D.2	Theorems and Proofs . . . . .	104

<b>E</b>	<b>Computer Codes</b>	<b>111</b>
E.1	Utility Routines . . . . .	112
E.1.1	Build Two-Link Arm System: buildtla.m . . . . .	112
E.1.2	Build Double Integrator System: buildddin.m . . . . .	115
E.1.3	Check Dimensional Consistency: check.m . . . . .	118
E.1.4	Install Gains in System: update.m . . . . .	120
E.1.5	Find Phase of $\Psi$ Discontinuities: cases.m . . . . .	120
E.1.6	Display Time and Date: shotime.m . . . . .	123
E.2	Analysis Routines . . . . .	124
E.2.1	Top-level Structure . . . . .	124
E.2.2	User Interface: analyze.m . . . . .	124
E.2.3	Main Program: mras.m . . . . .	125
E.2.4	STM Calculation: psix.m . . . . .	128
E.3	Design Routines . . . . .	132
E.3.1	Top-level Structure . . . . .	132
E.3.2	User Interface: design.m . . . . .	132
E.3.3	Main Optimization Program: var_opt2.m . . . . .	133
E.3.4	Asynchronous Cost and Gradient: acstgrd2.m . . . . .	136
E.3.5	Asynchronous Cost Only: acst2.m . . . . .	140
E.3.6	Cost and Gradient Definition: cstgrd2.m . . . . .	143
E.3.7	STM and Partial: psi2plus.m . . . . .	145
E.3.8	Precomputation for psi2plus: trains2.m . . . . .	153
E.3.9	STM Only: psi2.m . . . . .	158
E.3.10	Linear Search: linsrch2.m . . . . .	160
	<b>Bibliography</b>	<b>164</b>

# List of Tables

8.1	Parameters for Double Integrator Examples . . . . .	60
8.2	Parameters for Two-Link Arm Examples . . . . .	71

# List of Figures

2.1	Simultaneous Independent Controllers . . . . .	8
2.2	Superposition of Simultaneous Independent Sample Processes . . . . .	9
2.3	Asynchronous BTP and Phase . . . . .	11
3.1	Phase Concept . . . . .	21
3.2	Block Diagram of Hybrid System . . . . .	22
3.3	Event Timeline Illustration . . . . .	25
3.4	Timeline Illustration . . . . .	26
4.1	Synchronous Stability Plot, Nominal Gains. . . . .	37
4.2	Asynchronous Stability Plot, Nominal Gains. . . . .	38
5.1	Block Diagram for Example System . . . . .	40
5.2	Stability Plot, Nominal Gains, Long BTP . . . . .	42
8.1	Block Diagram for Double Integrator System . . . . .	60
8.2	Step Response: Nominal Gains, Synchronous Sampling . . . . .	61
8.3	Step Response: Nominal Gains, Asynchronous Sampling . . . . .	62
8.4	Synchronous Stability Plot, Synchronous Gains . . . . .	63
8.5	Step Response: Synchronous Gains, Synchronous Sampling . . . . .	63
8.6	Asynchronous Stability Plot, Synchronous Gains . . . . .	64
8.7	Step Response: Synchronous Gains, Asynchronous Sampling . . . . .	64
8.8	Asynchronous Stability Plot, Asynchronous Gains . . . . .	65
8.9	Asynchronous Stability Plot, Asynchronous Gains, Long BTP . . . . .	66
8.10	Step Response: Asynchronous Gains, Asynchronous Sampling . . . . .	66

8.11 Synchronous Stability Plot, Asynchronous Gains . . . . .	67
8.12 Step Response: Asynchronous Gains, Synchronous Sampling . . . . .	67
8.13 Optimal Cost as a Function of Sample Rate . . . . .	68
8.14 Block Diagram for Two-Link Robot . . . . .	71
8.15 Synchronous Stability Plot, Synchronous Gains . . . . .	72
8.16 Step Response: Synchronous Gains, Synchronous Sampling . . . . .	73
8.17 Step Response: Synchronous Gains, Asynchronous Sampling . . . . .	74
8.18 Asynchronous Stability Plot, Asynchronous Gains . . . . .	74
8.19 Step Response: Asynchronous Gains, Asynchronous Sampling . . . . .	75
8.20 Step Response: Asynchronous Gains, Synchronous Sampling . . . . .	76
C.1 Phase Relationships . . . . .	100



# Chapter 1

## Introduction

### 1.1 Purpose

The purpose of this research was to develop methods to design and analyze asynchronous digital control systems. Digital control systems can provide high reliability with low cost and weight using modern digital integrated circuits. Distributed system architecture can also reduce cost or improve fault tolerance. Also, unsynchronized distributed systems may be easier to build. Consequently, asynchronous digital design is a natural approach for implementing many distributed control systems.

Synchronized multirate systems are a special case of the asynchronous digital system. Synchronous multirate systems are periodically time varying because the sample process repeats exactly after some period. Existing multirate design approaches use this periodic behavior as the basis for design and analysis.

With true asynchronous systems, the ratio of sample periods is irrational and the sampling pattern never repeats exactly. However, asynchronous sample patterns can be approximated by synchronous patterns just as irrational numbers can be approximated by rational fractions to any desired accuracy. A similar approximation is the basis of the asynchronous design method developed here.

## 1.2 Related Literature

The asynchronous discrete-time control problem is the general case of the synchronous multirate digital control problem. Walton [Wal81] and Glasson [Gla83] developed comprehensive surveys of existing multirate methods. Multirate digital design has been a topic of continuing interest since the early 1950's. Existing multirate methods generally fall into two classes: frequency domain methods and time domain methods. All previous methods were limited to synchronous systems. Specifically, all sample period ratios had to be rational numbers such that the system was periodically time varying.

The original synchronous frequency domain techniques were Frequency Decomposition of Sklansky [SR55] and Switch Decomposition of Kranc [Kra57]. Subsequent efforts by Coffey and Williams [CW66], Jury [Jur68] [Jur67], Boykin and Frazier [BF75], and Whitbeck [WH78] extended and expanded the Switch and Frequency decomposition techniques.

Kalman and Bertram [KB59] described a general state space analysis technique for hybrid linear systems. This technique was the basis for several optimal control design approaches. Glasson and Broussard [GB79] [BH84] [BG80], Amit and Powell [Ami80] [AP81], and Lennartson [Len86] solved the optimal synchronous multirate state-feedback regulator problem. Berg and Powell [Ber86] solved the more general constrained problem for a controller of specified form (such as partial state feedback with constant gains).

All of these methods required synchronous sampling (rational sample period ratios).

## 1.3 Overview

This report addresses two related, multirate digital control topics. First, a sufficient stability criterion is developed for asynchronous multirate linear systems. Second, a new synchronous multirate design method is developed and extended to asynchronous systems.

The analysis approach described here was based on existing synchronous methods with an additional allowance to account for the difference between the actual sample pattern and a synchronous approximation. The resulting stability criterion is a sufficient condition that can guarantee asynchronous stability (but does not prove instability). The criterion also gives a figure of merit analogous to a rightmost  $s$ -plane bound for the poles of a linear, time-invariant system. This author doubts the existence of a necessary and sufficient condition for true asynchronous stability. In practice, this is not a problem because the synchronous approximation can be arbitrarily precise.

The design approach is an extension of the Constrained Optimization Synthesis design method [Ber86]. The cost and gradient formulations are new. A gradient search finds controller coefficients which minimize a scalar cost function. The designer specifies the controller structure, cost weighting, and the process noise. Then the algorithm finds optimal gains that minimize the weighted mean square state errors. If the specified process noise exceeds the synchronous approximation error the resulting design should be stable and well behaved.

This method and the Constrained Optimization Synthesis method produce identical results given the same synchronous problem. This new method has three advantages. First, this method avoids numerical overflow and manual fine-tuning during the gradient search. Second, generalized forms for the cost and process noise accommodate discrete measurement noise and saturation penalties. Finally, this method computes the gradients and Hessian only for actual feedback gains (instead of all possible feedback paths). This last factor gives a significant reduction in computation time and data storage.

The basic design method finds optimal feedback gains for synchronous sampling at a specified phasing (for instance, two samplers with simultaneous initial samples). This basic method is extended rigorously to synchronous sampling with random phasing. Without rigorous proof, heuristic arguments are made for applying the extended method to the true asynchronous sampling case. The stability of the resulting asynchronous design can be verified with the asynchronous stability criterion, and the performance can be evaluated by simulation.

PC-MATLAB [MLBK85] computer codes were developed for the design and analysis methods. These codes are included in Appendix E. Sample design cases illustrate the methods and some properties of asynchronous systems.

## 1.4 Scope of Methods

The design and analysis methods are suitable for multi-input/multi-output systems. The formulation does not distinguish between plant and controller or between states, inputs, or controls. The methods apply to linear systems composed of:

- a linear, time-invariant continuous-time part described by  $\dot{x}_c = Ax_c + Bu$  where  $A$  and  $B$  are constant matrices,  $x_c$  is the continuous state vector, and  $u = x_s$ .
- sample-and-hold elements with outputs  $x_s$  described by a finite number of state transition matrices of the form  $x_s(t^+) = S_i X(t^-)$  where each  $S_i$  is constant and  $X = [x_c^T x_s^T x_d^T]^T$ , and
- a discrete-time part described by a finite number of different state transition matrices of the form  $x_d(t^+) = D_i X(t^-)$  where each  $D_i$  is constant.

The usual continuous measurement and feed-forward matrices ( $C$  and  $D$ ), typical of the general linear, time-invariant system, are embedded in the  $S_i$  and  $D_i$  matrices. Any number of arbitrary, periodic sample schedules may simultaneously control the discrete time events ( $S_i$  and  $D_i$ ). The main restriction in both the analysis and design methods is that the closed loop state transition matrix (STM) must have a full set of eigenvectors (non-defective). However, the design method extends to defective STM with repeated eigenvalues at zero.

## 1.5 Contributions

The primary contributions of this research are:

- The sufficient stability criterion for asynchronous digital systems.

- An improved constrained optimization synthesis method for synchronous multirate designs.
- Extension of the synchronous multirate method to the synchronous case with random phasing and to the asynchronous case.

## Chapter 2

# Problem Description

This section defines synchronous and asynchronous sampling and discusses reasons for employing asynchronous designs. The merits of fast and slow sample rates are discussed. Finally, several current design approaches and their limitations are reviewed.

### 2.1 Motivation

Multirate digital control design is motivated, primarily, by plants with a wide range of natural frequencies. For example, if the plant has a 100 Hz mode and a 1 Hz mode, the fast modes may require 300 to 1,000 control updates per second while 3 to 10 updates per second are adequate for the 1 Hz mode. Using the same high rate for both modes wastes computer capacity and measurement/control signal bandwidth. Hence, multirate control is a natural approach for many real applications.

The usual motivation for asynchronous control is a distributed control system architecture with multiple controllers. This situation may occur when several cheap computers are used instead of one expensive computer or when the controllers are not physically colocated. In either case, synchronizing all the sample processes to the same master clock may increase cost and complexity. When asynchronous designs give adequate performance, the additional cost and complexity is unnecessary.

System integrity requirements may also motivate asynchronous design. In a

fault-tolerant design, the plant may have several independent controllers, each capable of controlling the plant with the others failed. In this case, requiring synchronized controllers may introduce a single-point failure mode.

## 2.2 Asynchronous Definitions

The problem is to devise methods to design and analyze systems with asynchronous digital controllers. Key asynchronous concepts and definitions are now reviewed to help clarify and focus the problem.

### 2.2.1 Simultaneous Sample Processes

Asynchronous systems have two or more independent digital controllers operating simultaneously as shown in Figure 2.1. The relative timing between the sample processes will be called phase (precise definition later). The overall sequence of events, or sample schedule, is found by superimposing events from the independent processes as shown in Figure 2.2.

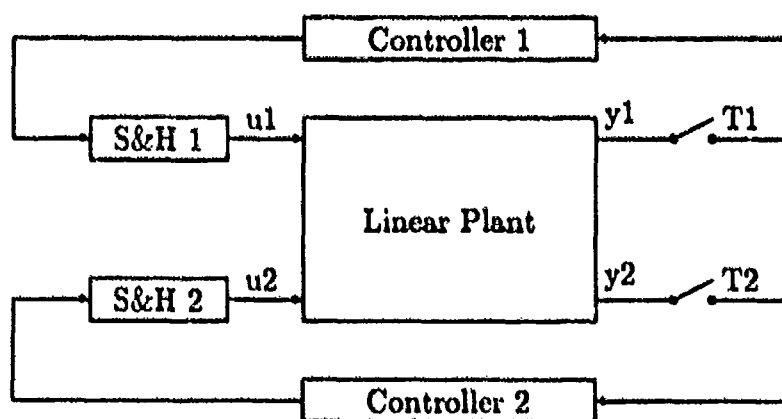


Figure 2.1: Simultaneous Independent Controllers

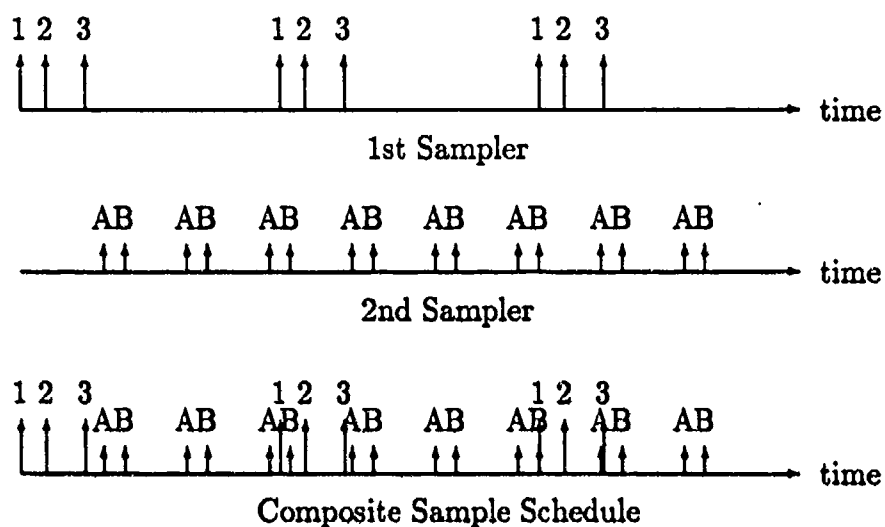


Figure 2.2: Superposition of Simultaneous Independent Sample Processes

### 2.2.2 Synchronous and Asynchronous Sampling

The intuitive idea of asynchronous sampling (independent samplers with no master clock) is essentially correct. In practice, independent clocks do not stay synchronized exactly. Therefore, even identical samplers will drift in “phasing” when controlled by independent clocks. A rigorous definition for asynchronous and asynchronous sampling follows.

Consider a system with  $n$  periodic sampling processes with periods  $T_1, T_2, \dots, T_n$ . If there exists a set of positive integers  $\{k_1, k_2, \dots, k_n\}$  such that  $T_1 k_1 = T_2 k_2 = \dots = T_n k_n$ , then the system is synchronous. Alternatively, the ratio of every two sample periods is a rational number for synchronous systems. Synchronous systems are periodic with finite period (period =  $T_i k_i$ ).

Conversely, asynchronous systems have at least one pair of sample periods whose ratio is irrational. With asynchronous sampling, the sample schedule is not periodic.



### 2.2.3 Basic Time Period

Observe that if the set  $\{k_1, k_2, \dots, k_n\}$  satisfies the above condition for synchronous sampling, then  $\{jk_1, jk_2, \dots, jk_n\}$ , with  $j$  a positive integer, also satisfies the condition. A unique basic time period (BTP) for synchronous systems is defined as  $T_i k_i$  where  $\{k_1, k_2, \dots, k_n\}$  have no common factors.

A non-unique basic time period (BTP) can be defined for asynchronous systems as follows. Suppose the designer selects a set of positive integers  $\{k_1, k_2, \dots, k_n\}$  such that:  $T_1 k_1 \approx T_2 k_2 \approx \dots \approx T_n k_n$ . Then  $T_i k_i$  is a BTP and the  $i$ 'th sample sequence is designated the key sequence (since the BTP is keyed to that sequence). Note that there are an infinite number of choices for the  $k$ 's. The  $T_i k_i$  products can be made arbitrarily close by choosing large  $k$ 's (long BTP's). This is analogous to using fractions to approximate irrational numbers.

### 2.2.4 Phase

For a given BTP, the phase of a sequence is the elapsed time from the start of the BTP to the initial discrete event of the sequence. By convention, every sequence starts with a discrete event. The key sequence phase is defined to be zero so all phases are unique.

Let  $T_i$  be the period of the key sequence and  $T_x$  be the period of some other sequence: "x". If  $T_i k_i \neq T_x k_x$ , phase for sequence "x" will be different in successive BTP's. If sequence "i" and sequence "x" are asynchronous, then  $\frac{T_x}{T_i}$  is irrational and sequence "x" has a different phase in every BTP. Finally, if  $\frac{T_x}{T_i}$  is irrational, the phase of sequence "x" is assumed to be uniformly distributed on  $[0, T_x)$  when all future BTP's are considered.

Figure 2.3 illustrates these ideas for the two-sequence case where  $T_1 = \pi$ ,  $T_2 = 1$ ,  $k_1 = 1$ , and  $k_2 = 3$ . The phase of the asynchronous sequence is  $\tau$ . Note that each BTP has a unique  $\tau$  and each  $\tau$  uniquely defines the subsequent  $\tau$ .

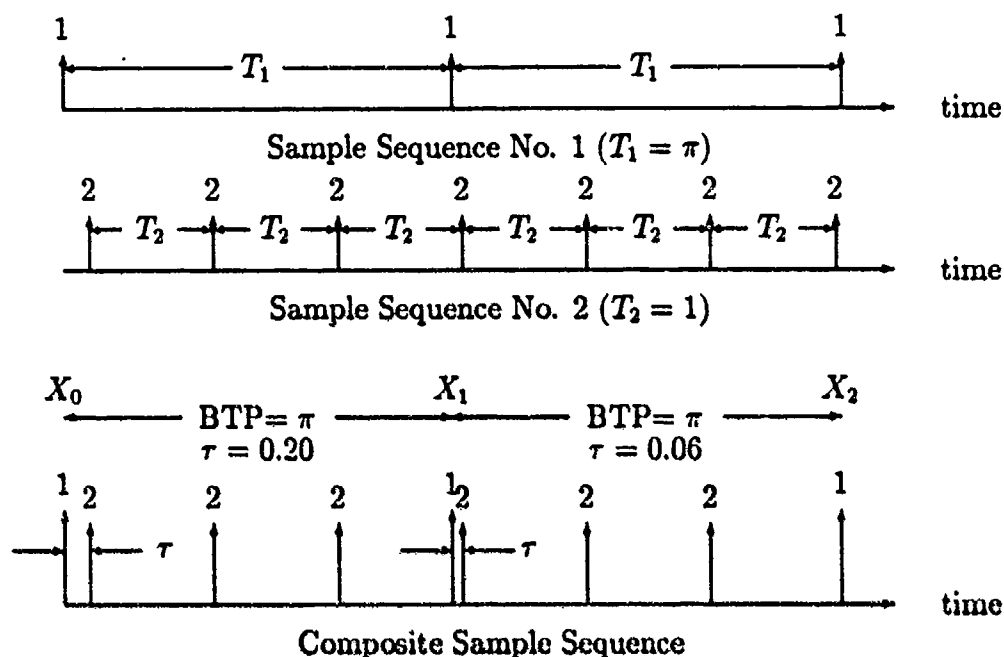


Figure 2.3: Asynchronous BTP and Phase

### 2.2.5 BTP Segments and Events

Each BTP consists of periods of continuous state propagation separated by discrete state updates (e.g. sample-and-hold updates). Each period of continuous state propagation with constant discrete states is called a segment. Each discrete state ( $x_s$  or  $x_d$ ) update is called an event.

## 2.3 Sample Rate Considerations

Many current designs use high sample rates. Such sample rates are much higher than the minimum rates needed to satisfy performance specifications. In some cases (such as manual control systems) other considerations mandate high rates. In other cases, good performance can be achieved with lower sample rates.

One reason for using high sample rates is that a fast digital controller closely approximates a continuous controller. The error between a true analog controller and the digital approximation is, effectively, process noise. Consequently, any robust

analog design can be simulated by a digital controller with high sample rates. Also, the asynchronous sampling issue can be safely ignored because all control loops are essentially continuous (with process noise) and asynchrony is a discrete-time phenomenon.

High sample rates have drawbacks. The most obvious is the cost and complexity of fast control computers, fast analog/digital converters, and wide-band data paths. Roundoff error accumulation can be a subtle problem, particularly if complex processing is used. Finally, considerable filtering may be needed to estimate rate information because the parameter change between measurements is small compared to roundoff error (high noise to signal ratio).

High sample rates may require higher analog/digital conversion precision to control roundoff accumulation and to reduce process noise correlation time. This combination of high data rates and long data words compounds the hardware performance requirements.

## 2.4 Limitations of Existing Design Methods

Highlights of several representative multirate design methods will be reviewed with emphasis on the limitations, design difficulty, and implementation difficulty.

### 2.4.1 Ad Hoc Approaches

First, here are several common ad hoc approaches. These "methods" often work (i.e. produce acceptable performance) when the sample rate is high, but they lack a rigorous theoretical foundation. These methods tend to break down when slow sampling is used and when the different sample rates are nearly equal. These are not true design methods. They are actually methods of approximating the multirate problem as a series of single-rate approximations which can be solved by existing single rate methods. The design difficulty and controller complexity are determined by the underlying single-rate method.

### Discretized Analog Design

This approach starts with a design for a satisfactory analog controller. Then, a "digital equivalent" design is obtained from the analog controller design (typically by one of the methods in [FP80]). This approach can always be used and cross-coupling may be exploited to advantage (if the analog design method has this property). However, the resulting controller may be unsatisfactory. In fact, it is easy to construct simple examples where adequate analog controllers have unstable digital equivalents.

### One Loop at a Time

This approach requires the designer to associate groups of outputs with groups of inputs according to the plant modes they affect. To illustrate, the fastest modes are identified along with inputs that can control those modes and outputs where those modes are observable. Then, a single rate controller is designed for this input/output/mode group using some standard single rate method. The controller for the fast inner loop now becomes part of the plant and the process continues until all loops are closed and all modes are controlled.

If all sample rates are integer multiples of the next slower sample rate and all samplers are synchronized, this is the exact one loop at a time method discussed in the next section. Otherwise, analog approximations of each succeeding controller must be used to define each "new" analog plant containing embedded digital inner-loop controllers.

This method is very general and can always be used. However, the best grouping of inputs and outputs may be unclear. Furthermore, coupling between mode controls is one-way in the sense that slow modes are designed considering the fast modes (which are now part of the plant) but not vice-versa. Consequently, the fast modes may not compensate for shortcomings in the slow modes. Finally, the assumption that the fast inner loops are "almost continuous" is valid only if each inner loop sample rate is much faster than the next outer loop. In short, the resulting design may work well or it may be unstable; simulations are needed to validate

the results.

### Singular Perturbation Method

The singular perturbation concept for ignoring minor effects has been rigorously studied [Kok84]. The method discussed here employs the general singular perturbation idea without rigorous analysis.

This is similar to the "one loop at a time method" but in the reverse order. The fast modes should be much faster than the slow modes. If the fast modes are well controlled (closed loop) then the fast modes can be assumed in constant equilibrium with the slow modes when the slow modes are analyzed. Conversely, the slow modes are assumed constant when the fast modes are being analyzed. Stated another way, the fast mode transient dies so quickly that the slow modes do not respond to it; hence, the mode groups are effectively decoupled.

This method designs the slow controller first based on the assumption that the fast modes are in constant equilibrium. Then a fast controller is designed based on the partially closed-loop system including the slow controller. For the sampled data case, the fast controller is designed assuming the slow modes (or the slow controls at least) are constant. This is clearly an approximation in the discrete-time case.

The advantages and disadvantages of this method are essentially the same as for the one loop at a time method except that the cross coupling is reversed. This reversal is usually good since fast modes are better able to compensate errors in slow modes than the other way around. The approach is limited to cases where the plant modes are well separated in frequency.

#### 2.4.2 Exact Methods

These methods are rigorous for true lumped, linear time-invariant plants. Hence, controllers designed with these methods should perform as predicted (except for plant model error). However, all these methods are limited to the synchronous sampling case.

### One Loop at a Time

This method, discussed earlier, becomes exact when each sample rate is an integer multiple of the next slower rate and the samplers are synchronized. In this special case, the continuous plant can be transformed into a true discrete equivalent model and the design can be done entirely in the Z-domain. All the earlier comments concerning this method apply except the model becomes exact (analog approximation is eliminated) and the sample rates must be synchronized integer multiples. The design difficulty and controller complexity are determined by the method used to design each loop.

### Frequency Domain Methods

The original multirate methods were the Switch Decomposition technique of Kranc [Kra57] and Frequency Decomposition technique of Sklansky [SR55]. Jury [Jur68] later showed these techniques to be equivalent.

**Switch Decomposition** For synchronous sampling, the sample process repeats each BTP. The switch decomposition formulation uses samplers which all operate simultaneously at the BTP rate with advances ( $\exp(sT_i)$ ) and delays ( $\exp(-sT_i)$ ) to shift the sample time and the control output times to the appropriate points within the BTP.

This is an exact analysis technique for synchronous multirate systems. It is primarily an analysis method. Therefore, it gives little guidance for choosing the controller structure or gains in the multi-input/multi-output case.

**Frequency Decomposition** The frequency decomposition method can be applied when all sample periods are integer multiples of some short time period. Discrete equivalents ( $T(z_n)$ ) of the continuous transfer functions are developed for sampling at the short time period. Then  $T(z_n^k)$  represents the discrete transfer function for a slower sampler with period "k" times the short time period.

This too is an exact analysis technique for synchronous multirate systems. Like switch decomposition, it gives little guidance for choosing the controller structure

or gains in the multi-input/multi-output case.

### Time Domain Methods

**Optimal Synthesis** Several investigators including Amit [Ami80], Glasson [GB79], and Lennartson [Len86] have devised different ways to solve this problem. This is an optimal regulator problem where state feedback gains are found to minimize a scalar cost function. The cost function is a standard Linear Quadratic Regulator (LQR) cost integral. The solution is found from solving the resulting time-varying periodic Riccati Equation. The resulting gains are periodic but different for each segment of the BTP.

Controllers designed by this approach should exhibit the harmonious control blending and robustness characteristics of other optimal regulators. Furthermore, the design process is automatic once the weights and process noise are specified. Recent work [Len86] claims efficient methods to solve the time-varying Riccati Equation.

The approach is limited to synchronous systems. Also, measurements of all states must be available. Finally, the resulting controller is relatively complex since it must schedule different gains for each segment in the Basic Time Period.

**Constrained Optimal Synthesis** Berg [Ber86] developed this approach to generalize the optimal synthesis design method. The same cost function is used; however, the designer specifies the form of the controller. The optimal gains for the specified controller structure are found by numerical optimization.

This method has many of the advantages of the Optimal Synthesis method with the added advantage of handling a variety of control structures. Hence, simple controllers with fixed gains and partial state feedback and controllers with dynamics can also be considered.

The method is limited to synchronous sampling. The added flexibility requires the designer to guess the best controller structure. Finally, the gradient search for optimal control coefficients (particularly in the original implementation) can be computationally intensive and requires frequent designer intervention to fine-tune

the convergence process.

### 2.4.3 Common Limitations

All of the exact methods require synchronous sampling. None of the methods have objective criteria for optimal sample rate selection. Only the Optimal Synthesis methods addressed finite word-length (quantization and roundoff error) and plant uncertainties (design robustness) by including process noise. None of the methods directly addressed system integrity with respect to failures.



## Chapter 3

# Definitions, Conventions, and Notation

This section addresses a variety of background items, definitions, and notation that provide a foundation for subsequent chapters. The system description nomenclature parallels the corresponding variables in the PC-MATLAB code (Appendix E).

### 3.1 Synchronous Sampling

Consider a system with  $n$  periodic sampling processes with periods  $T_1, T_2, \dots, T_n$ . If there exist a set of positive integers  $\{k_1, k_2, \dots, k_n\}$  such that

$$T_1 k_1 = T_2 k_2 = \dots = T_n k_n, \quad (3.1)$$

then the system is synchronous and periodic with period  $T_i k_i$ .

### 3.2 Basic Time Period

#### 3.2.1 Synchronous Basic Time Period

A unique basic time period (BTP) for synchronous systems is defined as  $T_i k_i$  where  $\{T_1, T_2, \dots, T_n\}$  and  $\{k_1, k_2, \dots, k_n\}$  satisfy Equation 3.1 and  $\{k_1, k_2, \dots, k_n\}$  have no common factors.

For synchronous sampling, the BTP is unique. A synchronous BTP is the shortest period such that the discrete event sequence is the same for all BTP's. The state transition matrix is the same for all synchronous BTP's.

### 3.2.2 Asynchronous Basic Time Period

A non-unique basic time period (BTP) for asynchronous systems is defined as  $T_i k_i$  where the designer selected a set of positive integers  $\{k_1, k_2, \dots, k_n\}$  such that:

$$T_1 k_1 \approx T_2 k_2 \approx \dots \approx T_n k_n.$$

Also, the  $i$ 'th sample sequence is defined as the key sequence (since the BTP is keyed to that sequence).

For asynchronous sampling, the designer specifies the BTP. The asynchronous BTP is not unique. The sequence and timing of the discrete events is never *exactly* the same for any two BTP's; however, the discrete event sequence and timing is nearly the same for each pair of sequential BTP's. The key sequence is always synchronized with the BTP.

### 3.2.3 Sequence Phasing

The phase ( $\tau$ ) of a sequence is defined as the elapsed time from the start of a BTP to the initial discrete event of the sequence. Defining key sequence phase to be zero makes the other phases unique. Figure 3.1 illustrates this concept.

Let  $T_i$  be the period of the key sample sequence and  $T_x$  be the period of some other sequence. Observe that phase for sequence "x" will be different in subsequent BTP's unless  $T_i k_i = T_x k_x$ . Furthermore, if the ratio  $T_x$  to  $T_i$  is irrational, then sequence "x" will have a different phase in every BTP. Finally, this phase is assumed to be uniformly distributed on  $[0 T_x)$  when sampling is asynchronous.

For a given BTP, the sequence and timing of the discrete events is uniquely described by a vector of phase times (one component for each sample sequence). By convention, the phase time for the key sequence is always zero. Hence, if there are  $n_s$  sample sequences, there are  $n_s$  phase times but one of these is zero.

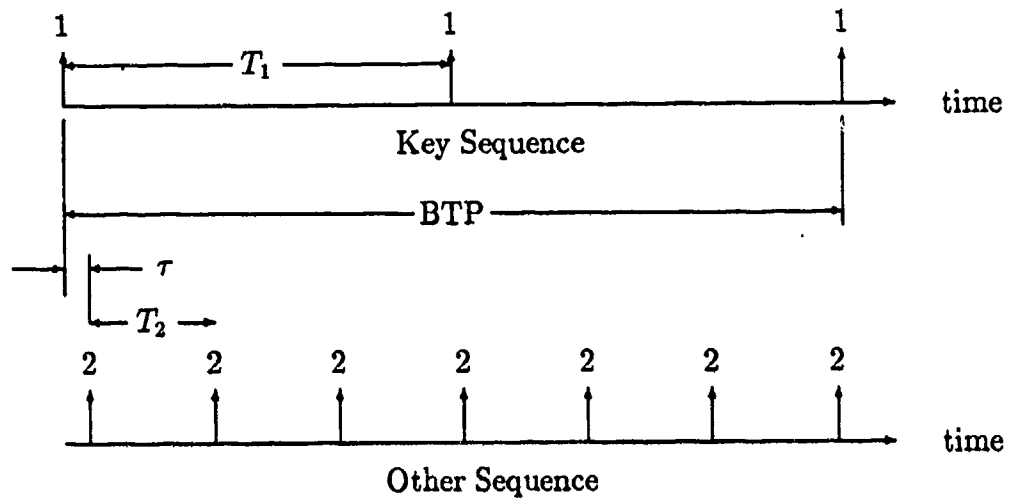


Figure 3.1: Phase Concept

### 3.3 Kalman-Bertram Representation

In [KB59], Kalman and Bertram present a systematic way to construct the state transition matrix (STM) for an arbitrary hybrid analog/digital system over an arbitrary time period. Part of their approach, which is used in this paper, will be summarized here.

#### 3.3.1 The State Vector

The state vector has three parts: continuous states, sample states, and discrete states. If the system were represented as a block diagram composed of integrators, sample-and-hold elements, and discrete delay elements, the integrator outputs are continuous states, the sample-and-hold outputs are sample states, and the delay element outputs are discrete states. Figure 3.2 represents the general hybrid system where all paths represent vector quantities. Integrator inputs can only be connected (through gains) to integrator outputs and sample-and-hold outputs because delay-element outputs are undefined between discrete events. The equations for the samplers include the usual continuous output and feedforward matrices ("C" and "D"). Also, the sampler equations must avoid any algebraic loops or the problem

will be ill posed. These three types of states represent all the memory in the system. Hence, the system state is:

$$X = \begin{bmatrix} x_c \\ x_s \\ x_d \end{bmatrix}$$

where:  $x_c$  is a column vector of continuous states,  $x_s$  is a column vector of sample states, and  $x_d$  is a column vector of discrete states.

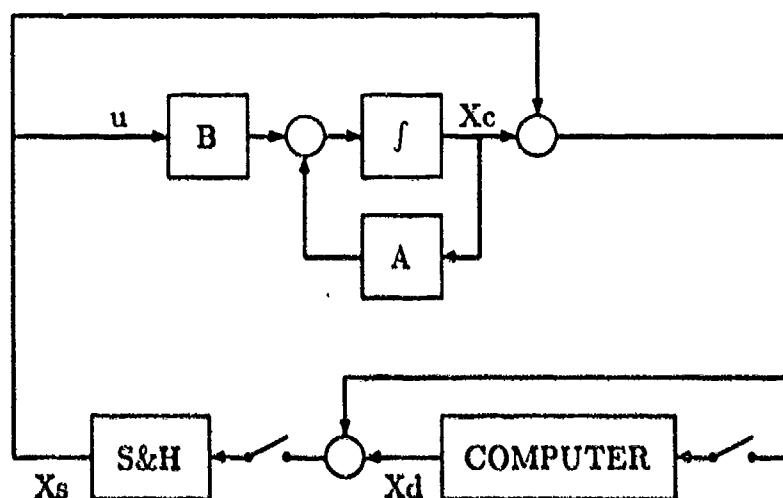


Figure 3.2: Block Diagram of Hybrid System

### 3.3.2 State Transition Matrices

There are two types of state transitions: continuous transitions and discrete transitions. Continuous transitions occur during segments and discrete events occur between segments.

### Continuous State Transitions

The continuous state transition can be computed for any time interval ( $\Delta t$ ) containing no sample or discrete transitions. Assume the continuous state space description:  $\dot{x}_c = Ax_c + Bu$  is known and note that  $u(t) = x_s$ , which is constant during  $\Delta t$ . Use the usual zero-order-hold continuous-to-discrete conversion (see [FP80] or [Kai80]) to obtain:

$$x_c(t_0 + \Delta t) = \phi(\Delta t)x_c(t_0) + \Gamma(\Delta t)u(t_0)$$

where  $\phi(\Delta t) = \exp(A\Delta t)$  and  $\Gamma(\Delta t) = \int_0^{\Delta t} \exp(As)B ds$ . Or:

$$x_c(t_0 + \Delta t) = \phi(\Delta t)x_c(t_0) + \Gamma(\Delta t)x_s(t_0).$$

Combining this with  $x_s(t_0 + \Delta t) = I_s x_s(t_0)$  and  $x_d(t_0 + \Delta t) = I_d x_d(t_0)$ , where  $I_s$  and  $I_d$  are identity matrices of the correct size, yields the full state transition matrix:

$$X(t_0 + \Delta t) = \begin{bmatrix} \phi(\Delta t) & \Gamma(\Delta t) & 0 \\ 0 & I_s & 0 \\ 0 & 0 & I_d \end{bmatrix} \begin{bmatrix} x_c(t_0) \\ x_s(t_0) \\ x_d(t_0) \end{bmatrix} = \Phi(\Delta t)X(t_0). \quad (3.2)$$

Given a fast and reliable matrix exponentiation routine [MVL78], a simple way to find  $\phi(\Delta t)$  and  $\Gamma(\Delta t)$  is given by [VL78]:

$$\begin{bmatrix} \phi(\Delta t) & \Gamma(\Delta t) \\ 0 & I_s \end{bmatrix} = \exp\left(\begin{bmatrix} A & B \\ 0 & 0 \end{bmatrix} \Delta t\right). \quad (3.3)$$

Another form, useful in derivations is:

$$\Phi(\Delta t) = \exp\left(\begin{bmatrix} A & B & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \Delta t\right) \quad (3.4)$$

### Sample State and Discrete State Transitions

Transition matrices for sample states can be found by inspection. For example, suppose the sample event consists of sample and hold state "k" sampling state "m" through a gain  $K_m$  and state "n" through a gain  $K_n$  (i.e.  $x_k(t^+) = K_m x_m(t^-) +$

$K_n x_n(t^-)$ ). The corresponding state transition matrix is just an identity matrix with the  $k$ 'th row replaced by the row vector:

$$\begin{bmatrix} 0 & \cdots & 0 & K_m & 0 & \cdots & 0 & K_n & 0 & \cdots & 0 \end{bmatrix}$$

where  $K_m$  is in the  $m$ 'th column and  $K_n$  is in the  $n$ 'th column.

Similarly, the discrete state transition matrices are obtained from the difference equations by inserting the difference equation coefficients into the corresponding rows of the transition matrix. The remaining rows of the matrix are the same as an identity matrix.

The transition matrices for sample and discrete state transitions are essentially similar and a single discrete-time transition often updates both sample states and discrete states. The main reason for differentiating between sample and discrete states is that sample states control the continuous states. Therefore, the sample states should be grouped together in the state vector so the columns of  $\Gamma$  stay together.

### 3.3.3 BTP State Transition Matrix

Given the discrete transition matrices ( $D_1, D_2, \dots, D_n$ ),  $A$ , and  $B$ , a state transition matrix for a BTP can be constructed for any specified sequence of events. To illustrate, suppose the BTP begins with a discrete event of type 1, followed by elapsed time  $t_1$ , followed by a discrete event of type 2, and finally an elapsed time  $t_2$ , completing the BTP (see Figure 3.3). The corresponding state transition matrix for this sequence is simply:

$$\Psi = \Phi(t_2)D_2\Phi(t_1)D_1.$$

## 3.4 Asynchronous System Description

We need a precise and compact notation to describe a hybrid (analog and discrete) system with arbitrary periodic sampling. The notation shown here parallels the PC-MATLAB code in Appendix E. The continuous-time (analog) part of the system is

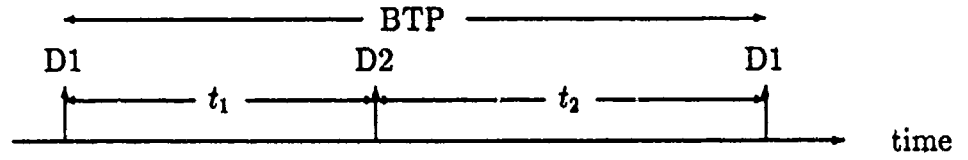


Figure 3.3: Event Timeline Illustration

completely specified by the  $A$  and  $B$  matrices of the matrix differential equation:  $\dot{x}_c = Ax_c + Bx_s$ . The columns of  $B$  are arranged so they are in one-to-one correspondence with the sample-and hold states. For example, a duplicate column is added for a particular input if two separate sample states drive the input. Conversely, columns for two different inputs might be summed into a single column if one hold state drives two inputs. For computational simplicity,  $A$  and  $B$  can be stored as a single square array (zero-filled at the bottom):

$$AB = \begin{bmatrix} A & B \\ 0 & 0 \end{bmatrix}.$$

Thus the full analog transition matrix can be evaluated as:

$$\Phi(\Delta t) = \begin{bmatrix} \exp(AB\Delta t) & 0 \\ 0 & I_d \end{bmatrix}.$$

Each sample schedule is finite and periodic so there are only a finite number of different discrete time transitions. In practice, discrete transitions often update both sample and discrete states, so no distinction is made between discrete-time transitions that update sample states and discrete-time transitions that update discrete states. The discrete transitions are numbered arbitrarily and the full state transition matrices can be stored as  $DS$ , a row of square block arrays:

$$DS = [ D_1 \ D_2 \ \dots \ D_n ].$$

The sample sequences are defined by a rectangular "sequence pointer" array which has a row for each sample sequence and enough columns to describe the

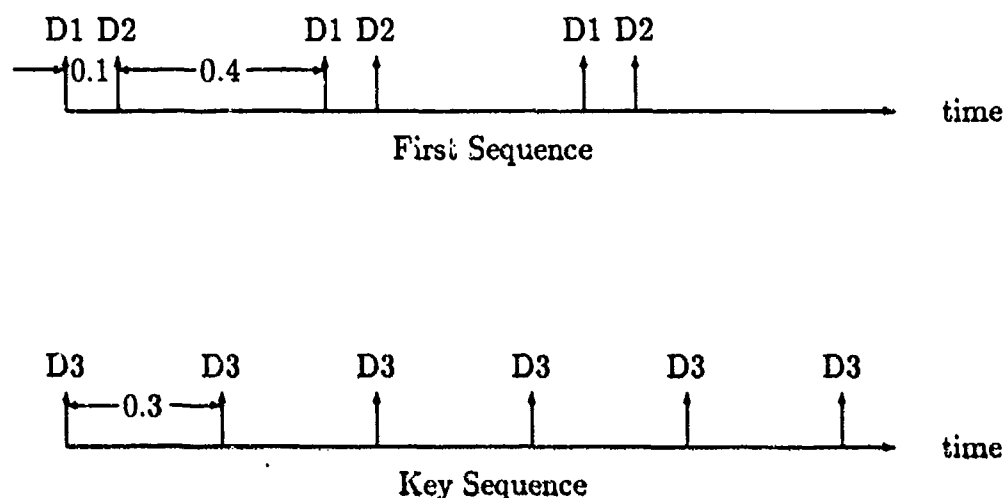


Figure 3.4: Timeline Illustration

longest sequence. Negative integers indicate discrete transitions defined by the corresponding block of the *DS* array. Non-negative real numbers indicate the duration of continuous-time transitions. When some sequences are longer than others, rows of the short sequences are filled with zeros (indicating a continuous transition for zero elapsed time: i.e. no transition). By convention, the first event in every sequence is a discrete event. To illustrate, consider the sequences sketched in Figure 3.4 described by the sequence pointer:

$$\text{seqptr} = \begin{bmatrix} -1 & 0.1 & -2 & 0.4 \\ -3 & 0.3 & 0 & 0 \end{bmatrix}.$$

This array indicates two simultaneous sequences. The first sequence has period 0.5 and consists of discrete event 1 followed by discrete event 2 after a delay of 0.1. The second sequence is just discrete event 3 repeated with period 0.3. Based on the preceding BTP discussion, this is a synchronous system with  $\text{BTP}=1.5$ .

### 3.5 Cost Function

A generalized Linear Quadratic Regulator (LQR) cost function is used in the design method. One would expect that the resulting designs share the desirable robust



properties of the LQR.

The cost function in [Ber86] was:

$$\tilde{J}(N) \equiv E \left\{ \frac{1}{2N PT} \int_0^{N PT} \begin{bmatrix} x(t) \\ u(t) \end{bmatrix}^T \begin{bmatrix} \tilde{Q}(t) & 0 \\ 0 & \tilde{R}(t) \end{bmatrix} \begin{bmatrix} x(t) \\ u(t) \end{bmatrix} dt \right\},$$

where  $PT$  is the BTP,  $E\{\cdot\}$  is the expected value operator, and  $\tilde{Q}$  and  $\tilde{R}$  are designer-specified weighting matrices. The initial state covariance was zero ( $E\{x(0)x^T(0) = 0\}$ ) and the analog process noise had covariance  $W(t)$ . If  $\tilde{Q}$ ,  $\tilde{R}$ , and  $W$  are either constant or periodic with period=BTP (as in [Ber86]), then as  $N \rightarrow \infty$  this cost becomes the steady-state mean square state error and control effort weighted by  $\tilde{Q}$  and  $\tilde{R}$ . In the limit, this is exactly equivalent to

$$\tilde{J} = E \left\{ \frac{1}{2PT} \int_0^{PT} \begin{bmatrix} x(t) \\ u(t) \end{bmatrix}^T \begin{bmatrix} \tilde{Q}(t) & 0 \\ 0 & \tilde{R}(t) \end{bmatrix} \begin{bmatrix} x(t) \\ u(t) \end{bmatrix} dt \right\},$$

if the process noise begins at  $t = -\infty$ . In this case,  $E\{x(0)x^T(0)\}$  is a steady state value at BTP start, not zero.

For the formulation in this paper, controls and states are combined in a single state vector. The above cost function is generalized to:

$$\begin{aligned} J &= E \left\{ \frac{1}{2BTP} \int_0^{BTP} \begin{bmatrix} x_c(t) \\ x_s(t) \\ x_d(t) \end{bmatrix}^T \begin{bmatrix} w_{11} & w_{12} & 0 \\ w_{21} & w_{22} & w_{23} \\ 0 & w_{32} & w_{33} \end{bmatrix} \begin{bmatrix} x_c(t) \\ x_s(t) \\ x_d(t) \end{bmatrix} dt \right\} \\ &= E \left\{ \frac{1}{2BTP} \int_0^{BTP} [X(t)]^T [W_0] [X(t)] dt \right\}, \end{aligned} \quad (3.5)$$

where  $W_0$  is a positive semi-definite cost weighting matrix specified by the designer. If state errors are equally serious throughout the BTP, a constant  $W_0$  is not a significant limitation. Choosing non-zero values for  $w_{31}$  and  $w_{13}$  makes the cost computation significantly more difficult. There is no apparent reason to specify non-zero values for  $w_{31}$  and  $w_{13}$ , so they are zero by convention and the PC-MATLAB computer codes in Appendix E do not include contributions from  $w_{31}$  and  $w_{13}$ . Conversely, non-zero values for  $w_{33}$ ,  $w_{12}$ , and  $w_{23}$  can be used to avoid saturation

and limits. Observe that  $w_{11} = \tilde{P}$  and  $w_{22} = \tilde{Q}$  so this formulation is exactly the same as [Ber86] when the other  $w_{ij}$  terms are zero. Finally,  $w_{22}$  should be positive definite to insure that infinite control commands are not allowed.

### 3.6 Process Noise

The process noise used here is also an extension of the continuous process noise used in [Ber86]. The process noise covariance is described by:

$$X^0 = \begin{bmatrix} x_{11}^0 & 0 & 0 \\ 0 & x_{22}^0 & x_{23}^0 \\ 0 & x_{32}^0 & x_{33}^0 \end{bmatrix}.$$

where  $x_{11}^0 = E\{w(t)w^T(t)\}$  is the continuous process noise covariance. This is the same as the process noise in [Ber86] when  $w(t)$  is stationary. The other terms:  $x_{22}^0$ ,  $x_{23}^0$ ,  $x_{32}^0$ , and  $x_{33}^0$  are discrete process noises covariances which are added to the state error covariance whenever the corresponding states are updated. To illustrate, suppose that state  $i$ , a sample state, and state  $j$ , a discrete state, are updated simultaneously by the same discrete state transition matrix; then,  $X^0(i, i)$ ,  $X^0(j, j)$ ,  $X^0(i, j)$ , and  $X^0(j, i)$  are added to the corresponding elements of the state error covariance matrix when the transition occurs. These additional terms can be used to specify digital measurement noise since all sampled measurements immediately become discrete or sample states.

### 3.7 Cost Function Scope

Before continuing, note that this extended cost function is quite general. The traditional Linear Quadratic Gaussian (LQG) cost function, with process noise and measurement noise, is fully accommodated within the framework of the extended cost function. Equivalent process noise (in the  $x_s$  and  $x_d$ ) can be found from discrete measurement noise. Hence, this formulation will accomplish LQG design if the controller structure is an observer with state feedback.

In addition, the off-diagonal  $W_0$  blocks allow penalties that can be used to avoid state and control limits. For example, suppose state  $j$  is limited and  $\dot{x}_j = x_k$  where  $x_k$  is a hold state (i.e. a control), then  $W_0(j, k) > 0$  penalizes control that pushes  $x_j$  toward the limit, particularly when  $x_j$  is large. The  $w_{23}$  block can be used in a similar way to avoid control limits.



## Chapter 4

# Asynchronous Stability Condition

This chapter develops the sufficient stability condition for an asynchronous sampled data system. Specifically, if the sum of two integrals is negative, the system is stable on the average. Furthermore, this sum defines an upper bound on the exponential decay constant for the average state error decay rate. An analogous figure of merit for linear, time invariant continuous systems is a rightmost bound on the pole locations in the s-plane. Finally, plots of the two integrands show how stability is influenced by asynchrony and phasing. Hence, the approach is also useful for synchronous systems with unknown phase. Although the method is applicable to any number of asynchronous sample schedules, a simple two-schedule (one asynchronous schedule) version is derived and used for the examples in the next chapter. The main limitation is that the closed-loop BTP state transition matrix (STM) must be diagonalizable.

### 4.1 Stability Definition

Consider a linear, time-varying system with a state vector  $x(t)$  and a state transition matrix  $\Psi(t_f, t_0)$  such that  $x(t_f) = \Psi(t_f, t_0)x(t_0)$ .

A necessary and sufficient condition for global asymptotic stability is:

$$\lim_{t_f \rightarrow \infty} \bar{\sigma}(\Psi(t_f, t_0)) \rightarrow 0, \text{ for all } t_0 \quad (4.1)$$

where  $\bar{\sigma}(\cdot)$  denotes the maximum singular value of a matrix. This stability definition will be used throughout this paper. Note that this says that each element in  $\Psi(\infty, t_0)$  must be zero.

## 4.2 Factors of $\Psi$

For reasons that will be apparent later,  $\Psi$  will first be factored in a certain way.

Divide the interval  $[t_f, t_0]$  into finite segments  $[t_n, t_{n-1}], \dots, [t_2, t_1], [t_1, t_0]$  such that  $n \rightarrow \infty$  as  $t_f \rightarrow \infty$ . Then,  $\Psi(t_f, t_0)$  in Equation 4.1 can be factored as

$$\Psi(t_f, t_0) = \Psi_n \Psi_{n-1} \cdots \Psi_2 \Psi_1$$

where  $\Psi_k$  is defined as  $\Psi(t_k, t_{k-1})$ . Later, these intervals will be set to the "basic time period" (BTP).

When the individual  $\Psi_k$ 's are diagonalizable (not defective) each  $\Psi_k$  can be factored ( $\Psi_k = S_k \Lambda_k S_k^{-1}$ ) where  $\Lambda$  is a diagonal matrix of eigenvalues and  $S$  is a matrix of eigenvectors. Then  $\Psi(t_f, t_0)$  can be expanded:

$$\Psi(t_f, t_0) = S_n \Lambda_n S_n^{-1} S_{n-1} \Lambda_{n-1} S_{n-1}^{-1} \cdots S_2 \Lambda_2 S_2^{-1} S_1 \Lambda_1 S_1^{-1}.$$

Finally, factor each

$$\Lambda_k = \sqrt{\frac{\Lambda_k}{\bar{\lambda}_k}} \bar{\lambda}_k \sqrt{\frac{\Lambda_k}{\bar{\lambda}_k}},$$

where  $\bar{\lambda}_k$  is the spectral radius of  $\Lambda_k$  or  $\Psi_k$ . Now:

$$\begin{aligned} \dots \Psi_k \Psi_{k-1} \dots &= \dots S_k \Lambda_k S_k^{-1} S_{k-1} \Lambda_{k-1} S_{k-1}^{-1} \dots \\ &= \dots S_k \sqrt{\frac{\Lambda_k}{\bar{\lambda}_k}} \bar{\lambda}_k \sqrt{\frac{\Lambda_k}{\bar{\lambda}_k}} S_k^{-1} S_{k-1} \sqrt{\frac{\Lambda_{k-1}}{\bar{\lambda}_{k-1}}} \bar{\lambda}_{k-1} \sqrt{\frac{\Lambda_{k-1}}{\bar{\lambda}_{k-1}}} S_{k-1}^{-1} \dots \\ &= \dots S_k \sqrt{\frac{\Lambda_k}{\bar{\lambda}_k}} \bar{\lambda}_k \Delta_k \bar{\lambda}_{k-1} \sqrt{\frac{\Lambda_{k-1}}{\bar{\lambda}_{k-1}}} S_{k-1}^{-1} \dots \end{aligned}$$

$$\text{where: } \Delta_k \equiv \sqrt{\frac{\Lambda_k}{\bar{\lambda}_k}} S_k^{-1} S_{k-1} \sqrt{\frac{\Lambda_{k-1}}{\bar{\lambda}_{k-1}}}. \quad (4.2)$$

So,  $\Psi(t_f, t_0)$  can be rewritten in factored form as:

$$\Psi(t_f, t_0) = S_n \sqrt{\frac{\Lambda_n}{\bar{\lambda}_n}} \bar{\lambda}_n \Delta_n \bar{\lambda}_{n-1} \Delta_{n-1} \cdots \bar{\lambda}_2 \Delta_2 \bar{\lambda}_1 \sqrt{\frac{\Lambda_1}{\bar{\lambda}_1}} S_1^{-1}. \quad (4.3)$$

### 4.3 General Sufficient Condition

Recalling that  $\bar{\sigma}(\cdot)$  is a scalar consider an equivalent form for Equation 4.1:

$$\lim_{t_f \rightarrow \infty} \bar{\sigma}(\Psi(t_f, t_0)) \rightarrow 0. \iff \lim_{n \rightarrow \infty} [\bar{\sigma}(\Psi_n \Psi_{n-1} \cdots \Psi_2 \Psi_1)]^{\frac{1}{n}} < 1$$

Furthermore

$$\begin{aligned} & \bar{\sigma}(\Psi_n \Psi_{n-1} \cdots \Psi_2 \Psi_1) \leq \\ & \bar{\sigma}\left(S_n \sqrt{\frac{\Lambda_n}{\bar{\lambda}_n}}\right) \bar{\lambda}_n \bar{\sigma}(\Delta_n) \bar{\lambda}_{n-1} \bar{\sigma}(\Delta_{n-1}) \cdots \bar{\lambda}_2 \bar{\sigma}(\Delta_2) \bar{\lambda}_1 \bar{\sigma}\left(\sqrt{\frac{\Lambda_1}{\bar{\lambda}_1}} S_1^{-1}\right). \end{aligned}$$

Therefore a sufficient stability condition is:

$$\lim_{n \rightarrow \infty} \left[ \bar{\sigma}\left(S_n \sqrt{\frac{\Lambda_n}{\bar{\lambda}_n}}\right) \bar{\lambda}_n \bar{\sigma}(\Delta_n) \bar{\lambda}_{n-1} \bar{\sigma}(\Delta_{n-1}) \cdots \bar{\lambda}_2 \bar{\sigma}(\Delta_2) \bar{\lambda}_1 \bar{\sigma}\left(\sqrt{\frac{\Lambda_1}{\bar{\lambda}_1}} S_1\right) \right]^{\frac{1}{n}} < 1. \quad (4.4)$$

Suppose we define bounds  $\lambda_i$  and  $\sigma_i$  and the  $\Delta$ 's and  $\bar{\lambda}$ 's are divided into classes such that  $\bar{\sigma}(\Delta) \leq \sigma_i$  for all the  $\Delta$ 's in class "i" and  $\bar{\lambda}(\Psi) \leq \lambda_j$  for all  $\bar{\lambda}(\Psi)$ 's in class "j". Also, for any sequence length  $n$  let  $n_i$  be the number of  $\Delta$ 's in class "i" and let  $n_j$  be the number of  $\bar{\lambda}$ 's in class "j". Then

$$\begin{aligned} & \left[ \bar{\sigma}\left(S_n \sqrt{\frac{\Lambda_n}{\bar{\lambda}_n}}\right) \bar{\lambda}_n \bar{\sigma}(\Delta_n) \bar{\lambda}_{n-1} \bar{\sigma}(\Delta_{n-1}) \cdots \bar{\lambda}_2 \bar{\sigma}(\Delta_2) \bar{\lambda}_1 \bar{\sigma}\left(\sqrt{\frac{\Lambda_1}{\bar{\lambda}_1}} S_1\right) \right]^{\frac{1}{n}} = \\ & \left[ \bar{\sigma}\left(S_n \sqrt{\frac{\Lambda_n}{\bar{\lambda}_n}}\right) \left(\prod_{k=1}^n \bar{\lambda}_k\right) \left(\prod_{k=2}^n \bar{\sigma}(\Delta_k)\right) \bar{\sigma}\left(\sqrt{\frac{\Lambda_1}{\bar{\lambda}_1}} S_1\right) \right]^{\frac{1}{n}} \leq \\ & \left[ \bar{\sigma}\left(S_n \sqrt{\frac{\Lambda_n}{\bar{\lambda}_n}}\right) \right]^{\frac{1}{n}} \prod_i \sigma_i^{\frac{n_i}{n}} \prod_j \lambda_j^{\frac{n_j}{n}} \left[ \bar{\sigma}\left(\sqrt{\frac{\Lambda_1}{\bar{\lambda}_1}} S_1\right) \right]^{\frac{1}{n}} \end{aligned}$$

In taking the limit as  $n \rightarrow \infty$ , the first and last terms (assumed finite) go to unity so a sufficient stability condition becomes:

$$\lim_{n \rightarrow \infty} \prod_i \sigma_i^{\frac{n_i}{n}} \prod_j \lambda_j^{\frac{n_j}{n}} = \prod_i \sigma_i^{p_{\sigma i}} \prod_j \lambda_j^{p_{\lambda j}} < 1 \quad (4.5)$$

where  $p_{\sigma i}$  and  $p_{\lambda j}$  are the probability of occurrence of state transition matrices with  $\Delta$  in class i and  $\bar{\lambda}$  in class j respectively.

## 4.4 Asynchronous Multirate Stability

Now, apply Equation 4.5 to the asynchronous multirate problem. Let each  $[t_k, t_{k-1}]$  interval be a basic time period (BTP) and use rectangles in phase space to partition the  $\Delta$ 's and  $\lambda$ 's into classes.

Assume that the phase for all BTP's from  $t = 0$  to  $t = \infty$  is uniformly distributed. Then, the probability of occurrence of an STM in a given class is equal to the size of the phase space rectangle divided by the size of the total phase space.

Let  $\tau$  be the phase vector that uniquely defines the BTP state transition matrix. Then, for the  $k$ 'th component,  $\tau_k \in [0, T_k)$ , where  $T_k$  is the period of the  $k$ 'th sequence (a non-synchronous sample sequence). Assume the  $\tau_k$ 's are independent. Then, the probability of occurrence of a sequence with  $\tau_k \in [t_1, t_2]$  is just  $\frac{t_2 - t_1}{T_i}$  assuming  $0 \leq t_1, t_2 \leq T_i$ .

If we take the natural logarithm of Equation 4.5 we get

$$\sum_i p_{\sigma_i} \log(\sigma_i) + \sum_j p_{\lambda_j} \log(\lambda_j) < 0, \quad (4.6)$$

another form of the sufficient stability condition.

Suppose we divide the  $\Delta$ 's and  $\bar{\sigma}(\Psi)$ 's into classes based on rectangles (in  $\tau$  space) of size  $d\tau$ . Say the  $i$ 'th class is defined as all state transition matrices with  $\tau_1 \in [\tau_{i1}, \tau_{i1} + d\tau)$ ,  $\tau_2 \in [\tau_{i2}, \tau_{i2} + d\tau)$ ,  $\dots$   $\tau_n \in [\tau_{in}, \tau_{in} + d\tau)$ .

Now, Equation 4.6 can be expressed as sums over the coordinate indices ( $i \dots j$ ):

$$\frac{d\tau^n}{\prod_{i=1}^n T_i} \left( \sum_{i=1}^{n_1} \dots \sum_{j=1}^{n_n} \log(\sigma_{i \dots j}) \right) + \frac{d\tau^n}{\prod_{i=1}^n T_i} \left( \sum_{i=1}^{n_1} \dots \sum_{j=1}^{n_n} \log(\lambda_{i \dots j}) \right) < 0 \quad (4.7)$$

where:

$n$  = number of asynchronous samplers,

$$n_i = \frac{T_i}{d\tau},$$

$\lambda_{i \dots j} = \sup(\bar{\lambda}(\Psi(\tau)))$ ,  $\tau_1 \in [(i-1)d\tau, id\tau]$ ,  $\dots$   $\tau_n \in [(j-1)d\tau, jd\tau]$ , and

$\sigma_{i \dots j} \equiv \sup(\bar{\sigma}(\Delta(\tau)))$ ,  $\tau_1 \in [(i-1)d\tau, id\tau]$ ,  $\dots$   $\tau_n \in [(j-1)d\tau, jd\tau]$ .



In the limit as  $d\tau \rightarrow 0$ , the sums in Equation 4.6 become integrals:

$$\sigma^0 \equiv \frac{1}{\prod_{i=1}^n T_i} \int_{\tau_1=0}^{T_1} \cdots \int_{\tau_n=0}^{T_n} (\log(\bar{\sigma}(\Delta(\tau))) + \log(\bar{\lambda}(\Psi(\tau)))) d\tau_1 \cdots d\tau_n < 0. \quad (4.8)$$

Also, define  $\sigma^* = \frac{\sigma^0}{BTP}$ . The significance of  $\sigma^*$  will be discussed later.

#### 4.4.1 An Important Special Case

Finally, consider the special case with only one asynchronous sampler. Then,  $\tau$  becomes a scalar and Equation 4.8 becomes:

$$\sigma^0 \equiv \frac{1}{T} \int_{\tau=0}^T \log(\bar{\sigma}(\Delta(\tau))) d\tau + \frac{1}{T} \int_{\tau=0}^T \log(\bar{\lambda}(\Psi(\tau))) d\tau < 0. \quad (4.9)$$

This form will be used for the sample cases later in this paper.

### 4.5 Eigenvector Scaling

The eigenvector matrix ( $S$ ) needed in the " $\Delta$ " calculation is not unique. In fact, the choice of eigenvectors changes  $\sigma(\tau)$ . Ideally, the length of the eigenvectors in  $S$  would be selected to minimize  $\sigma(\tau)$  and reduce the conservative nature of the results. The best  $S$  (which minimizes  $\sigma(\tau)$ ) is not obvious. Numerical stability is another consideration. We can improve the numerical properties by choosing  $S$  in balanced modal form such that the norms of the columns of  $S$  are the same as the norms of the corresponding rows of  $S^{-1}$ . In some of the example cases, the balanced modal form improved the condition number of  $S$  (and  $S^{-1}$ ) significantly compared to choosing eigenvectors of unit length. This balanced modal decomposition was implemented in the computer codes in Appendix E. In most cases, the result ( $\sigma^*$ ) was less conservative than results with unit eigenvectors.

### 4.6 Analysis of Result

Before leaving this topic, consider the significance of the Equation 4.8. Comparing this with Equations 4.4 and 4.5 one can conclude that, in some average fashion,

$\log(\bar{\sigma}(\Psi(t_0 + BTP, t_0))) \leq \sigma^0$ . Hence, if  $X_0$  is some initial state and  $X_1$  is the state one BTP later, then on the average,  $\|X_1\| \leq \|X_0\| \exp(\sigma^0)$ . Hence,  $\sigma^0$  is like a maximum (conservative bound) decay constant for one BTP.

The term "average" was used loosely in this discussion since it actually indicates a multiplicative average. To be more precise, if the initial state error is  $X(0)$ , then the residual error  $\|X(t)\|$  will be less than  $\|X(0)\| \exp(\sigma^* t)$  for most values of  $t \in [0, \infty)$ . So,  $\sigma^*$  can be thought of as an average worst case (conservative) decay constant per unit time.

Now consider the integrands of Equations 4.8 and 4.9. The BTP STM is an analytic function of phase except at a finite number of points (see Appendix C). Hence, the  $\lambda_i$ 's and  $\Delta$ 's are continuous functions of  $\tau$  except at a few known values. Therefore, the integrals can be computed numerically without difficulty.

Plots of the integrands of Equation 4.9 versus  $\tau$  give substantial insight into the system behavior. If a system were actually synchronous with random phase,  $\log \bar{\lambda}(\Psi(\tau))$  must actually be negative for all  $\tau$  for guaranteed stability. Thus, a plot of  $\log \bar{\lambda}(\Psi(\tau))$  versus  $\tau$  shows how sequence phasing influences the stability of a synchronous system. Similarly, a plot of  $\log \bar{\sigma}(\Delta(\tau))$  versus  $\tau$  shows the possible destabilizing effect of asynchronous sampling. Together, these plots show the relative stability contributions of transitions over the BTP (if repeated indefinitely) and the mismatch in going from one BTP to the next. If the mismatch is too destabilizing, longer BTP's (closer to synchronous) can be used to get a "sharper" sufficient stability criterion.

To illustrate, consider Figures 4.1 and 4.2. Figure 4.1 represents a synchronous system. For synchronous systems at the synchronous BTP,  $\log \bar{\sigma}(\Delta(\tau)) = 0$  for all phase values. The system is stable for some values of phasing (where  $\log(\bar{\lambda})$  is negative) but not for others. If Figure 4.1 represented an asynchronous system, the system would be stable overall (area below curve is greater than area above) but the errors would grow during some BTP's. Such behavior may be acceptable if these "unstable" BTP's are mixed with stable BTP's. Conversely, if there are many "unstable" BTP's in a row, the behavior is probably unacceptable. Figure 4.2 represents a stable asynchronous system. The stability of the  $\lambda$  exceeds the worst

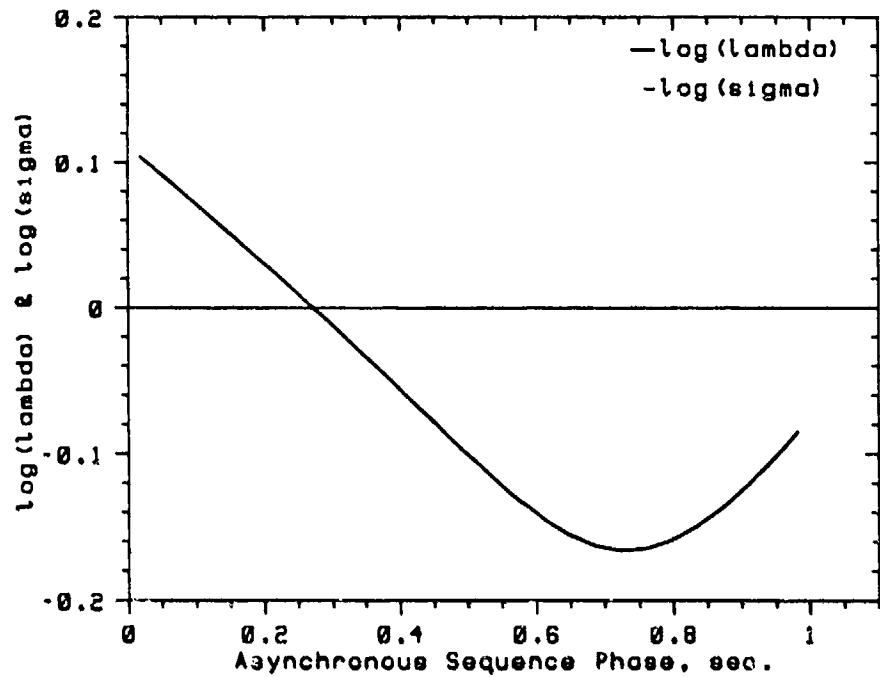


Figure 4.1: Synchronous Stability Plot, Nominal Gains.

possible destabilizing effect of  $\sigma$  (the BTP mismatch). Interestingly, a similar synchronous system is only stable for some  $\tau$  values.

One should not equate large negative values of  $\sigma^*$  with good performance. For example, lightly damped estimator error states can be perfectly acceptable when measurement noise is small. Increasing the damping for some state errors may only reduce damping of the output errors. Therefore,  $\sigma^*$  should only be used to assess stability, not performance.

The next Chapter applies these stability concepts to a simple system. For the remainder of this report, the following notational abbreviations will be used:

$$\lambda(\tau) \equiv \bar{\lambda}(\Psi(\tau)), \text{ and}$$

$$\sigma(\tau) \equiv \bar{\sigma}(\Delta(\tau)).$$

This yields a more compact notation and emphasizes the functional relationship between  $\lambda(\cdot)$ ,  $\sigma(\cdot)$ , and  $\tau$ .

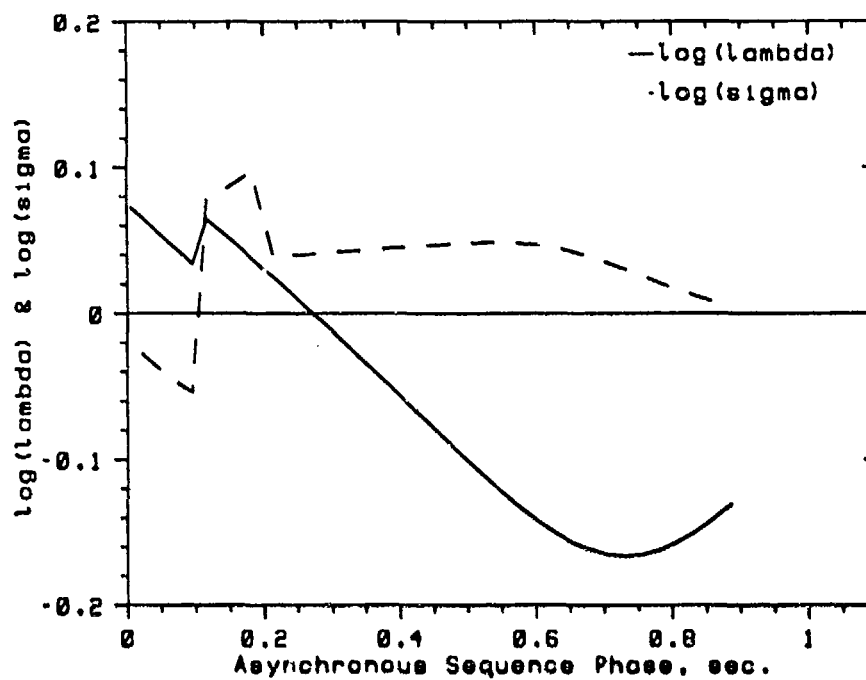


Figure 4.2: Asynchronous Stability Plot, Nominal Gains.

## Chapter 5

# Stability Examples

This chapter applies the sufficient stability condition to a simple double integrator plant. The results illustrate various stability considerations using this very simple example.

### 5.1 The System

The system consists of a double integrator plant with digital position feedback and a crude digital rate feedback (See Figure 5.1). The position and rate feedback represent two coupled but independent and possibly asynchronous sample processes. With analog state feedback, the system would be stable with any positive state feedback gains. The asynchronous digital system is fourth order and it is not so well behaved

The sample periods are  $T_1=T_2=1.0$  for synchronous cases and  $T_1=1$ ,  $T_2=0.9$  for asynchronous cases. The so-called asynchronous case is actually synchronous with a Basic Time Period (BTP) of 9.0. However, all cases will be analyzed with  $BTP = 1.0$ . This fiction allows direct comparison with an exact analysis using the true synchronous BTP.

This idea of using asynchronous analysis and a short BTP for a synchronous system has merit. For example, suppose some system were synchronous with  $BTP=100$  but we are interested in the response during a much shorter time (say 1). Then, it

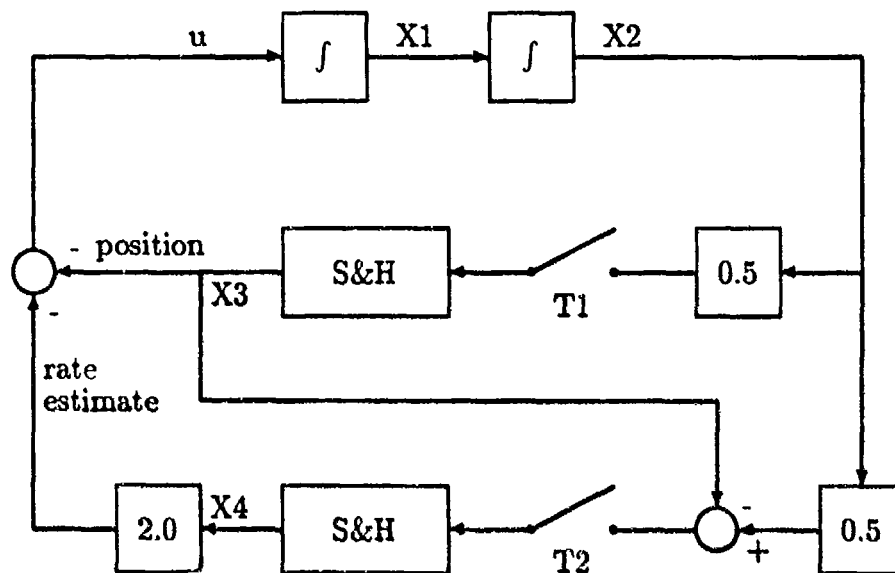


Figure 5.1: Block Diagram for Example System

would be appropriate to analyze the system with a small BTP (near 1) to evaluate the short-term stability of the system.

## 5.2 Nominal Sampling

At the nominal condition, both samplers operate at one sample per second. Figure 4.1 shows the stability condition (Equation 4.9) integrands versus phase for the nominal gains and synchronous sampling. The  $\sigma(\Delta)$  term is unity so  $\log \sigma(\Delta) = 0$  for all phase values. The  $\lambda$  term varies with phase. The system is stable at phase conditions greater than 0.28 and unstable for phase conditions less than 0.28. Therefore, the stability depends on the phase relationships between the two samplers. This is easy to understand from Figure 5.1. When the rate sampler operates just after the position sampler, the effective rate feedback gain is zero. As the rate sampler delay (phase) increases, the effective rate feedback gain increases and so does the

stability until the delay exceeds a quarter of the natural period where the effective rate feedback gain starts to decrease.

## 5.3 Asynchronous Sampling

### 5.3.1 Analysis for BTP=1

Now consider the case where the rate sampler is faster with a period of 0.9. Analyzing this configuration for the same BTP (1.0) gives  $\sigma^* = -0.031$  and the curves in Figure 4.2.

The  $\lambda$  curve is identical to the synchronous case for  $\tau > 0.1$ . The events in the asynchronous BTP are identical to the synchronous BTP whenever  $\tau$  is greater than 0.1; but the asynchronous BTP contains an extra rate sample when  $\tau$  is less than 0.1.

Since the sampling is actually synchronous, phase ( $\tau$ ) is not uniformly distributed. Actually,  $\tau$  cycles through a sequence of ten specific values (say 1.0, 0.9, 0.8, ... 0.2, 0.1). If the exact phasing is known, the actual values can be averaged. If the exact phasing is random, then  $\sigma^*$  is a good stability indicator for the synchronous case if the analysis BTP is much shorter than the true BTP.

### 5.3.2 Analysis for Synchronous BTP (9.0)

In fact, the so-called asynchronous example is actually synchronous. We use this fact to determine the true stability.

Figure 5.2 shows stability integrands for the true BTP. Comparing with the previous plot, we see the sufficient condition was conservative. This fact is, however, magnified by the crude synchronous approximation ( $1 \approx 0.9$ ). In actual practice, much better BTP's can be selected.

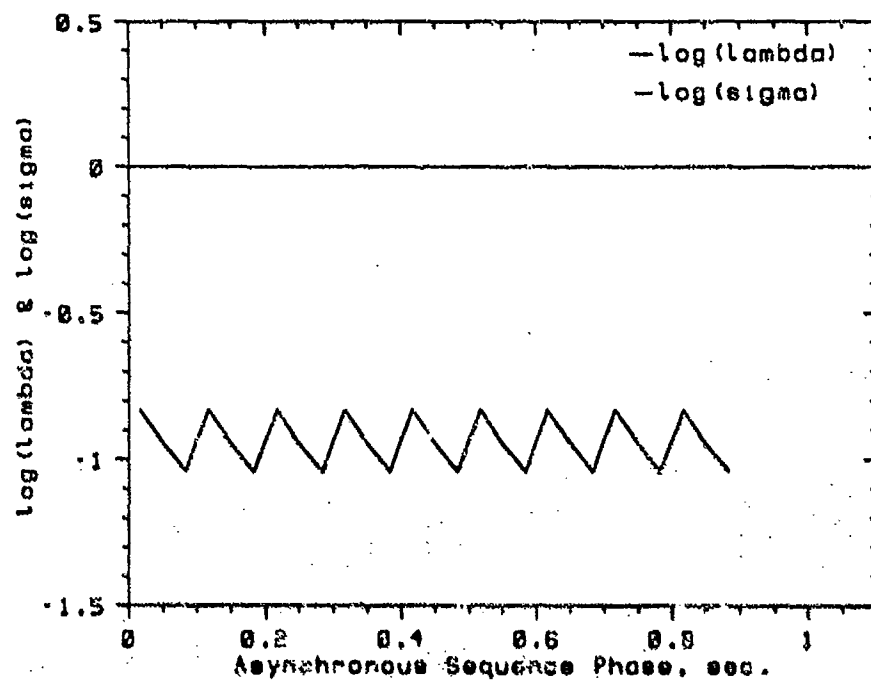


Figure 5.2: Stability Plot, Nominal Gains, Long BTP



## Chapter 6

# Synchronous Design Method

The overall design method is a gradient search to minimize a scalar cost function. This chapter describes the cost function, the gradient, and the search algorithm for the synchronous sampling case with known phase. The next chapter extends the method to synchronous sampling with random phase and to the asynchronous case with varying phase.

### 6.1 Cost and Gradient

Actually, two different cost functions are needed: one for an unstable system and one for a stable system. The true cost function must approach infinity as the system approaches instability because an unstable system is not acceptable (infinitely bad). However, initial guesses for the control gains may yield an unstable system. Therefore, an always finite, alternate cost function is used until stabilizing gains are found.

Ideally, the cost function is an analytic function of the BTP STM. At a minimum, the cost should be a continuous function of the gains with a gradient defined almost everywhere. Isolated gradient discontinuities are acceptable if the gradient search is not likely to encounter these points.

### 6.1.1 Initial Cost Function

The initial cost function must decrease as the system becomes more stable. An obvious choice is the spectral radius of the BTP STM since stability and a spectral radius less than one are equivalent. Unfortunately, the spectral radius gradient is discontinuous when multiple eigenvalues are equidistant from the origin. This is a serious flaw because a gradient search invariably encounters such conditions while minimizing the spectral radius of a high order system.

The initial cost function chosen is a variation of the  $L^n$  norm of the eigenvalue vector. The initial cost was defined as

$$L = \sum_{i=1}^k (\lambda_i^* \lambda_i)^{\frac{n}{2}} \quad (6.1)$$

where  $k$  is the order of the system,  $\lambda_i$  is an eigenvalue of the STM, and  $n$  is some even integer. As  $n$  increases, this cost function becomes equivalent to using the spectral radius. Typically, larger values of  $n$  are better for higher order problems. For the ninth-order two-link robot-arm problem,  $n = 4$  and  $n = 8$  both worked well. The gradient of this cost is defined whenever partials of the eigenvalues exist. Although the gradient is discontinuous at repeated roots, such points are rare and caused no problems for the test cases.

### 6.1.2 Initial Gradient

The gradient of the initial cost is found by applying the chain rule to Equation 6.1:

$$\dot{L} = \frac{n}{2} \sum_{i=1}^k (\lambda_i^* \lambda_i)^{\frac{n-2}{2}} (\dot{\lambda}_i^* \lambda_i + \lambda_i^* \dot{\lambda}_i). \quad (6.2)$$

The eigenvalue derivative ( $\dot{\lambda}_i$ ) is computed from  $\dot{\Psi}$  and  $\Psi$  as described in Appendix A. Gradient components are computed separately for each variable parameter in the controller.

### 6.1.3 Main Cost Function

Once a stable system is obtained, the cost function, as introduced in Chapter 3, is taken as:

$$J = E \left\{ \frac{1}{2 BTP} \int_0^{BTP} [X(t)]^T [W_0] [X(t)] dt \right\},$$

$$= E \left\{ \frac{1}{2 BTP} \int_0^{BTP} \begin{bmatrix} x_c(t) \\ x_s(t) \\ x_d(t) \end{bmatrix}^T \begin{bmatrix} w_{11} & w_{12} & 0 \\ w_{21} & w_{22} & w_{23} \\ 0 & w_{32} & w_{33} \end{bmatrix} \begin{bmatrix} x_c(t) \\ x_s(t) \\ x_d(t) \end{bmatrix} dt \right\}$$

for a system disturbed by stationary white process noise for the infinite past.

**Useful Theorems** The following theorems will be used to develop the cost function expressions (see Appendix D for proofs):

**Theorem 1** Let  $x_{cs}$  be the partial state vector:  $[x_c^T x_s^T]^T$  such that

$$\dot{x}_{cs} = [AB] x_{cs} + n_{cs}$$

where  $n_{cs}$  is a white, continuous process noise with covariance:

$$E\{n_{cs}(s)n_{cs}^T(t)\} = \begin{bmatrix} x_{11}^0 & 0 \\ 0 & 0 \end{bmatrix} \delta(s-t) = X_{cs}^0 \delta(s-t)$$

where  $\delta(\cdot)$  is a unit impulse at zero.

If  $xx(t) \equiv E\{x_{cs}(t)x_{cs}^T(t)\}$  and there are no discrete transitions between  $t = 0$  and  $t = t_1$ , then

$$xx(t_1) = \phi(t_1)xx(0)\phi(t_1)^T + R_c(t_1)$$

Where  $\phi(s) \equiv \exp([AB]s)$  and

$$R_c(t_1) = \int_0^{t_1} \phi(s)X_{cs}^0\phi(s)^T ds. \quad (6.3)$$

**Theorem 2** Let  $X = [x_c^T x_s^T x_d^T]^T$  be the full state vector. Let  $\Psi_d$  be a discrete state transition matrix where

$$X(t^+) = \Psi_d X(t^-) + n_d(t)$$

where  $n_d(t)$  is white discrete process noise with covariance  $R_d$ .

If  $XX(t) \equiv E\{X(t)X^T(t)\}$  and the discrete transition  $\Psi_d$  occurs at time  $t_0$ , then

$$XX(t_0^+) = \Psi_d XX(t_0^-) \Psi_d^T + R_d. \quad (6.4)$$

Note: let  $R_d = [R_{ij}]$ . Then  $R_{ij}$  is zero unless  $\Psi_d$  updates state  $i$  or state  $j$ .

**Theorem 3** Let  $X$  be the full state vector:  $[x_c^T x_s^T x_d^T]^T$  and

$$\dot{X} = G(t)X + N(t).$$

Where  $G(t)$  is stable with a corresponding state transition matrix  $\Psi(t_b, t_a)$ .  $N(t)$  is white process noise and  $E\{N(r)N^T(s)\} = \delta(r-s)R_n(s)$ .

Let  $XX(t) \equiv E\{X(t)X^T(t)\}$ ,  $t_0 > t_{-1} > t_{-2} \dots$ , and  $\lim_{n \rightarrow -\infty} (t_n) = -\infty$ .

Then if  $R_i \equiv E\{X(t_i)X^T(t_i)\}$  when  $E\{X(t_{i-1})X^T(t_{i-1})\} = 0$  (i.e. the covariance growth from  $t_{i-1}$  to  $t_i$ ) then

$$XX(t_0) = \sum_{i=0}^{-\infty} \Psi(t_0, t_i) R_i \Psi(t_0, t_i)^T.$$

**Theorem 4** Let  $X$  be the state vector:  $[x_c^T x_s^T x_d^T]^T$  such that

$$\dot{X} = GX + N$$

where  $G$  is constant and  $N$  is white, stationary process noise with covariance

$$X_c^0 = \begin{bmatrix} x_{11}^0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Let  $XX(t_i) \equiv E\{X(t_i)X^T(t_i)\}$  and  $W_0$  be a symmetric matrix.

If there are no discrete transitions between time=0 and time=t, then

$$\begin{aligned} J(t, 0) &\equiv E \left\{ \int_0^t X^T(r) W_0 X(r) dr \right\} \\ &= \tilde{\Sigma} \left( X X(0) \star \int_0^t \Psi(r)^T W_0 \Psi(r) dr \right) \\ &\quad + \tilde{\Sigma} \left( X_c^0 \star \int_0^t \int_0^r \Psi^T(s) W_0 \Psi(s) ds dr \right) \end{aligned} \quad (6.5)$$

where:  $J(t, 0)$  is the cost contribution for the segment  $(0, t)$ , " $\star$ " denotes element-by-element matrix multiplication,  $\tilde{\Sigma}$  denotes the algebraic sum of the matrix elements,

and  $\Psi(t) \equiv \exp \left( \begin{bmatrix} A & B & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} t \right).$

**Cost Calculation** The BTP consists of continuous time segments separated by discrete events. The discrete events do not contribute directly to the cost integral because they occur in zero time and the states are always finite. Hence, the cost is  $\frac{1}{2BTP} \sum_{i=1}^k J(t_i, t_{i-1})$  where  $J(m, n)$  is defined in Theorem 4 and there are  $k$  continuous segments in the BTP. Consider the second term in Theorem 4:

$$\tilde{\Sigma} \left( X_c^0 \star \int_0^t \int_0^r \Psi^T(s) W_0 \Psi(s) ds dr \right)$$

This term, which represents the cost contribution from the noise in the current segment, does not depend on the discrete (DS) gains (the other term represents the cost from all previous noise). For simplicity, this second term will be deleted entirely from the cost calculations. The rationale is:

- The value of the deleted term may be computed [VL78] but this is, by far, the most difficult calculation and its value is slight, at most.
- If all the controls are digital (all adjustable gains in DS array), then the deleted term is constant and it does not influence the gain optimization.
- With some analog controls (adjustable gains in the AB array) the contribution of this term will be small unless the fastest sample period is long compared to the error decay time constant (unlikely).

Using this simplification, the cost can be approximated as:

$$\frac{1}{2BTP} \sum_{i=1}^k \tilde{\Sigma} X X(t_i) \star \int_{t_i}^{t_{i+1}} \Psi(t)^T W_0 \Psi(t) dt$$

or

$$\frac{1}{2BTP} \sum_{i=1}^k \tilde{\Sigma} X X(t_i) \star Q_i$$

where

$$Q_i \equiv \int_{t_i}^{t_{i+1}} \Psi(t)^T W_0 \Psi(t) dt. \quad (6.6)$$

Notice that  $Q_i$  is always defined regardless of system stability.

The  $XX(t_i)$  term can be expanded using Theorem 3:

$$XX(t_i) = \Psi_{i0} X X(t_0) \Psi_{i0}^T + \sum_{j=1}^{i-1} \Psi_{ij} R_j \Psi_{ij}^T$$

where  $R_j$  is the additional state covariance from the end of the  $j - 1$ 'th segment to the end of the  $j$ 'th segment and  $\Psi_{mn}$  is the STM from the end of the  $n$ 'th continuous segment to the start of the  $m$ 'th continuous segment and  $n = 0$  indicates the start of the BTP.

Then, the cost becomes:

$$J = \frac{1}{2BTP} \sum_{i=0}^k \tilde{\Sigma} (\Psi_{i0} X X(t_0) \Psi_{i0}^T) \star Q_i + \frac{1}{2BTP} \sum_{i=1}^k \tilde{\Sigma} \left( \sum_{j=1}^{i-1} \Psi_{ij} R_j \Psi_{ij}^T \right) \star Q_i.$$

Now, use the symmetric identity:  $\tilde{\Sigma}(T X_1 T^T) \star X_2 \equiv \tilde{\Sigma}(T^T X_2 T) \star X_1$  to extract the  $XX(t_0)$  from the summation. The first term can be rewritten as:

$$\frac{1}{2BTP} \tilde{\Sigma} \left[ X X(t_0) \star \sum_{i=1}^k \Psi_{i0}^T Q_i \Psi_{i0} \right] = \frac{1}{2BTP} \tilde{\Sigma} X X(t_0) \star WW$$

where

$$WW \equiv \sum_{i=1}^{k-1} \Psi_{i0}^T Q_i \Psi_{i0}. \quad (6.7)$$

Notice that  $WW$  is always defined.

Suppose the additional covariance from noise in the current BTP is accumulated recursively in  $XX$  as follows:

$$XX_0^- = 0,$$

$$XX_i^+ = \Psi_{id} XX_i^- \Psi_{id}^T + W_{id} \text{ (reference theorem 2), and}$$

$$XX_i^- = \Psi_{ic} XX_{(i-1)}^+ \Psi_{ic}^T + \int_{t_i}^{t_{i+1}} \Phi(t) W_{oc} \Phi(t)^T dt \text{ (see theorem 1).}$$

$$XX = XX_k^-.$$

Where  $\Psi_{ic}$  is the  $i$ 'th continuous STM in the BTP,  $\Psi_{id}$  is the subsequent discrete STM, and there are  $k$  continuous segments in the BTP. The subscripts on  $X$  refer to the discrete events between continuous segments, and the  $(- +)$  superscripts indicate values just before and just after the indicated discrete events. These  $XX$  values will be used to simplify the cost expression. Note that the  $XX$  sequence can always be calculated.

Combining these expressions gives:

$$J = \frac{1}{2BTP} \bar{\Sigma}[XX(t_0) * WW] + \frac{1}{2BTP} \sum_{i=2}^k \bar{\Sigma}[XX_{i-1}^+ * Q_i]. \quad (6.8)$$

Expressions for each of these terms have been defined except  $XX(t_0)$  which is addressed now.

$XX$  is the state covariance at the end of a BTP if the state is zero at the BTP start.  $XX(t_0)$  is the steady state covariance at the start of the BTP if the system has been excited by the process noise forever. Note that  $XX$  always exists but  $XX(t_0)$  only exists if the system is stable. If the system is unstable, the steady state covariance is infinite and the initial cost function must be used.

For synchronous periodic sampling all BTP STM's ( $\Psi$ ) are identical. Using Theorem 3, the steady-state state covariance at BTP start ( $E\{X(t_0)X^T(t_0)\}$ ) can be found from  $XX$  and the BTP STM. When the BTP STM can be diagonalized, the infinite sum is solved in closed form.

$$\begin{aligned} XX(t_0) &= \sum_{i=0}^{\infty} \Psi(t_0 - t_i) XX_i \Psi^T(t_i - t_0) \\ &= \sum_{i=0}^{\infty} \Psi^i XX (\Psi^i)^T. \end{aligned} \quad (6.9)$$

$$\begin{aligned}
&= S \left( \sum_{i=0}^{\infty} \Lambda^i (S^{-1} X X S^{-H}) (\Lambda^H)^i \right) S^H. \\
&= S \left[ (S^{-1} X X S^{-H}) \star \left( \sum_{i=0}^{\infty} (\tilde{\Lambda}^i \tilde{\Lambda}^H)^i \right) \right] S^H. \\
&= S \left[ (S^{-1} X X S^{-H}) \star (\tilde{I} \div (\tilde{I} - \tilde{\Lambda} \tilde{\Lambda}^H)) \right] S^H.
\end{aligned}$$

Where:

$S, \Lambda$  = matrices of the eigenvectors and eigenvalues of  $\Psi$  where  $\Lambda$  is diagonal and  $\Psi = P \Lambda P^{-1}$ ,

$\tilde{\Lambda}$  is a column vector of the eigenvalues in  $\Lambda$ ,

$\tilde{I}$  is a matrix of all 1's,

$\star$  denotes element-by-element matrix multiplication, and

$\div$  denotes element-by-element matrix division.

Note that the quantity  $XX(t_0)$  always exists for a stable system; however, the closed form expression of Equation 6.9 only applies to stable systems where the BTP STM is diagonalizable. As a small, but useful, extension, if the STM cannot be diagonalized because of repeated defective eigenvalues at zero, switching to  $BTP' = k \cdot BTP$  (where  $k$  = number of roots at zero) will yield a diagonalizable STM.

An alternate method of finding  $XX(t_0)$  is to solve the Liapunov equation

$$XX(t_0) = \Psi XX(t_0) \Psi^T + XX.$$

Which merely states that  $XX(t_0)$  is the steady state covariance at the start of each BTP.

**Recursions** Fortunately, the preceding expressions lend themselves to orderly evaluation through a series of recursive relationships. Accumulators are established for  $XX$ ,  $WW$ ,  $\Psi$ , and  $YY$ ; where  $\Psi$  accumulates the BTP STM and  $YY$ , a scalar, accumulates the second term in Equation 6.8. Starting at the first event or segment in the BTP, with the corresponding state transition matrix ( $\psi$ ), the state covariance



growth  $R$ , and the cost weighting integral  $Q$  are computed for the event or segment. Then, the accumulators are updated using the recursion. This process continues til all segments and discrete events are included. Then  $XX(t_0)$  is found from Equation 6.9 (which uses the modal decomposition of  $\Psi$ ) and cost is found from Equation 6.8.

The initial conditions are:  $\Psi = I$ ,  $XX = 0$ ,  $WW = 0$ , and  $YY = 0$ . The recursions are:

$$YY = YY + \tilde{\Sigma}XX \star Q, \quad (6.10)$$

$$XX = \psi XX \psi^T + R, \quad (6.11)$$

$$WW = WW + \Psi^T Q \Psi, \text{ and} \quad (6.12)$$

$$\Psi = \psi \Psi. \quad (6.13)$$

Note that the  $YY$  update must precede the  $XX$  update and the  $WW$  update must precede the  $\Psi$  update. Also,  $Q$  is defined by Equation 6.6 for analog transitions and  $Q$  is zero for discrete transitions. The  $R$  term is the right hand term in Equation 6.3 for analog transitions and the right hand term in Equation 6.4 for discrete transitions. Notice that these recursions are valid and stable for stable and unstable systems and for defective BTP STM's. Therefore a-priori stability knowledge is not required. The appropriate cost function can be selected after the  $\Psi$ ,  $WW$ ,  $XX$ , and  $YY$  are computed and the spectral radius of  $\Psi$  is known. Appendix B shows how  $Q$ ,  $R$ , and  $\Psi$  are computed for continuous segments.

#### 6.1.4 Gradient of Main Cost Function

Define  $\dot{(\ )}$  as the partial of  $(\ )$  with respect to any of the adjustable parameters. When the gradient is required,  $\dot{YY}$ ,  $\dot{XX}$ ,  $\dot{WW}$ , and  $\dot{\Psi}$  are computed in parallel with  $YY$ ,  $XX$ ,  $WW$ , and  $\Psi$ . Initial conditions for  $\dot{YY}$ ,  $\dot{XX}$ ,  $\dot{WW}$ , and  $\dot{\Psi}$  are all zero. Parallel recursions for  $\dot{YY}$ ,  $\dot{XX}$ ,  $\dot{WW}$ , and  $\dot{\Psi}$  are obtained directly by differentiating Equations 6.10, 6.11, 6.12, and 6.13:

$$\dot{YY} = \dot{YY} + \tilde{\Sigma}(\dot{XX} \star Q + XX \star \dot{Q}), \quad (6.14)$$

$$\dot{XX} = \dot{\psi} XX \psi^T + \psi \dot{XX} \psi^T + \psi XX \dot{\psi}^T + \dot{R}, \quad (6.15)$$

$$\dot{W}W = \dot{W}W + \dot{\Psi}^T Q \Psi + \Psi^T \dot{Q} \Psi + \Psi^T Q \dot{\Psi}, \text{ and} \quad (6.16)$$

$$\dot{\Psi} = \dot{\psi} \Psi + \psi \dot{\Psi}. \quad (6.17)$$

Here too, the recursions must be performed in the given order. The expressions for  $XX(t_0)$  and cost are also differentiated to yield:

$$\begin{aligned} \dot{X}X(t_0) = & \dot{S}[T1 \star T2]S^H \\ & + S[\dot{T}1 \star T2]S^H \\ & + S[T1 \star \dot{T}2]S^H \\ & + S[T1 \star T2]\dot{S}^H. \end{aligned} \quad (6.18)$$

Where:

$$T1 = S^{-1} X X S^{-H},$$

$$\dot{T}1 = \dot{S}^{-1} X X S^{-H} + S^{-1} \dot{X} X S^{-H} + S^{-1} X X \dot{S}^{-H},$$

$$\dot{S}^{-1} = -S^{-1} \dot{S} S^{-1},$$

$$T2 = \tilde{I} \div (\tilde{I} - \tilde{\Lambda} \tilde{\Lambda}^H), \text{ and}$$

$$\dot{T}2 = (\dot{\tilde{\Lambda}} \tilde{\Lambda}^H + \tilde{\Lambda} \dot{\tilde{\Lambda}}^H) \star T2 \star T2.$$

Finally, the cost expression is differentiated to yield:

$$\dot{J} = \frac{1}{2BTP} (\tilde{\Sigma} (\dot{X}X(t_0) \star WW + XX(t_0) \star \dot{W}W) + \dot{Y}Y) \quad (6.19)$$

These partials are formed for each variable parameter. The eigenderivatives  $\dot{\Lambda}$  and  $\dot{S}$  are computed from the BTP STM ( $\Psi$ ) and its partial ( $\dot{\Psi}$ ) (see Appendix B).

## 6.2 Search Algorithm

The gradient search process begins by guessing values for the variable gains and installing them in the  $AB$  and  $DS$  matrices. Next, the BTP STM ( $\Psi$ ), the related parameters ( $WW$ ,  $XX$ , and  $YY$ ) and their partials (with respect to each variable

gain) are computed for the specified phasing. The main cost or initial cost function is selected based on the STM spectral radius. The cost and gradient are computed. A Quasi-Newton search is performed using the gradient and the Hessian. The Hessian is initialized at identity and updated with the Broyden-Goldfarb-Fletcher-Shanno update. Finally, the minimum along the search direction is found by a line search using a parabolic curve fit guarded by the golden-section step. These optimization methods are described in [GMW81].

The computer code for the search algorithm assumes that the system is unstable when random initial gains are used and it queries the operator if stored gains were used. When initial gains are unknown, random values may be used; but all-zero gains should be avoided. Setting the initial gains to zero may produce a rare singular condition where the gradient is not defined. When the system is initially assumed unstable, the algorithm switches to the main cost function when the BTP STM spectral radius drops below a threshold (e.g. 0.9). When a search step produces a marginally stable or unstable system after main cost function is selected, the cost routine returns a large positive number (1/eps was used). The system is treated as unstable whenever  $\max(\bar{\lambda}) > 0.99$ . Numerical overflow was never a problem.

The actual search algorithm code (see Appendix E) was a direct implementation of the equations given in [GMW81].



## Chapter 7

# Asynchronous Design Method

The vector space of all possible phase values can be divided into a finite number of regions where the BTP STM is an analytic function of phase (reference Appendix C). In fact, the STM, its eigensystem, and the related quantities ( $WW$ ,  $XX$ , and  $YY$  in the previous chapter) are polynomials of phase in each of these regions. If these regions are sufficiently small, a second order polynomial accurately describes the relation between cost and phase. This polynomial approximation gives a simple way of addressing the synchronous problem with unknown phase and, through that, the asynchronous problem.

### 7.1 Synchronous with Random Phase

Suppose we have a system with two independent (not synchronized) sample processes with synchronous periods. When the system is turned on, some random phase is established between the sample processes and that phase remains constant. Hence, the system is synchronous but the phase is random. Assume that the initial phase,  $\tau$ , is uniformly distributed on  $[0, T)$ . Let  $J(\theta, \tau)$  be the cost function of the previous chapter, where  $\theta$  is a vector of the variable parameters.  $J(\theta, \tau)$  is a computable scalar that is uniquely defined for each  $\theta, \tau$  pair. As before, the cost function is:

$$E \int_0^{BTP} X^T(t) W_0 X(t) dt = \frac{1}{T} \int_0^T J(\theta, \tau) d\tau$$

where the second integrand represents the expectation taken over  $\tau$ .

Direct numerical integration is difficult because each  $J(\theta, \tau)$  evaluation requires a great deal of computation. As an alternative, the region  $[0, T)$  can be divided into regions where  $J(\theta, \tau)$  is continuous. Then  $J(\theta, \tau)$  can be found for a few points in each region and  $J(\theta, \tau)$  can be modeled as a polynomial in  $\tau$  for that region. This polynomial is easily integrated and the gradient of the integral is easily found from the gradients of  $J(\cdot)$  at the same sample points.

### 7.1.1 One Random Phase

The curve-fit approach applies to  $\tau$ 's of any dimension, but the rest of this treatment will deal with the simple case of one sequence with random phase. From here on,  $\tau \in [0, T)$  is a scalar. In this case, there is a set  $\{t_0, t_1, \dots, t_k\}$  with  $t_i \in [0, T]$  and  $i = 1, \dots, k$  such that  $J(\theta, \tau)$  is continuous for  $\tau \in (t_i, t_{i+1})$ . Therefore,  $J(\cdot)$  will be approximated as a parabola in each region. Note that  $J(\theta, \tau)$  may be ill defined at  $\tau = t_i$  (a point of possible discontinuity).

#### Cost and Gradient

Let  $J_1$ ,  $J_2$ , and  $J_3$ , represent the cost:  $J(\theta, \tau)$  evaluated at  $\tau_1$ ,  $\tau_2$ , and  $\tau_3$ , where:

$$\tau_2 = \frac{t_i + t_{i+1}}{2}, \quad (7.1)$$

$$\tau_1 = t_i + \frac{\varepsilon}{2}(t_{i+1} - t_i), \quad (7.2)$$

and

$$\tau_3 = t_{i+1} - \frac{\varepsilon}{2}(t_{i+1} - t_i) \quad (7.3)$$

where  $\varepsilon$  is a safety factor to insure that the end points are actually in the current interval. In the interval  $(t_i, t_{i+1})$ , the cost function is be modeled as:

$$J(\theta, x) = cx^2 + bx + a$$

where  $a = J_2$ ,  $b = (J_3 - J_1)/2$ , and  $c = (J_3 - 2J_2 + J_1)/2$ . For simplicity,  $\tau$  is replaced by the scaled and zero-shifted dummy variable  $x$ , where  $x = 0$  corresponds

to  $\tau = \tau_2$ ,  $x = -1$  corresponds to  $\tau = \tau_1$ , and  $x = +1$  corresponds to  $\tau = \tau_3$ . Using this approximation, the contribution to the cost integral is:

$$\int_{t_i}^{t_{i+1}} J(\theta, \tau) d\tau \approx (t_{i+1} - t_i)(cx^2/3 + a) \Big|_{x=\frac{1}{1-\epsilon}}. \quad (7.4)$$

Likewise, for each gradient component:

$$\dot{J}(\theta, x) = \dot{c}x^2 + \dot{b}x + \dot{a}.$$

where  $\dot{a} = \dot{J}_2$ ,  $\dot{b} = (\dot{J}_3 - \dot{J}_1)/2$ , and  $\dot{c} = (\dot{J}_3 - 2\dot{J}_2 + \dot{J}_1)/2$ . So

$$\int_{t_i}^{t_{i+1}} \dot{J}(\theta, \tau) d\tau \approx (t_{i+1} - t_i)(\dot{c}x^2/3 + \dot{a}) \Big|_{x=\frac{1}{1-\epsilon}}. \quad (7.5)$$

Finally, the total estimated cost and gradient are found by summing the contributions from each region and dividing by  $T$ .

### 7.1.2 Design Algorithm

The design approach is a variation of the method for synchronous systems with known phase. As before, the first step is to guess initial values for the variable control gains. Now, however, the range of possible phase values is divided into continuous regions. The synchronous cost and gradient are computed at the center and near the ends of each region using exactly the same method as before. Then Equations 7.4 and 7.5 are used to estimate the cost and gradient in each continuous region for the random phase problem. The contributions from each interval are summed and the gradient search proceeds as before. In summary, the key difference is that a composite cost and a composite gradient are used instead of the cost and gradient (for one specified phase) derived in the last chapter. The composite cost and gradient for random phase are estimated from several values of the cost and gradient at specified phase values.

For the random-phase problem, there will always be at least one continuous region; but there may be many regions if the discrete transition matrices do not commute and the BTP contains multiple discrete transitions from different sample sequences. Therefore, many cost evaluations ( $J(\theta, \tau)$ ) may be required to evaluate a single composite cost and gradient.

Cost function selection (main or initial) is similar to the method with known phase except the switch from the initial to the main cost cannot be made until all sample points (say at  $\tau_1$ ,  $\tau_2$ , and  $\tau_3$ ) represent stable systems. The code implements the switch one gradient search after all points met the switch criteria. This one-step-late approach was necessary to avoid storing the STM data ( $\Psi$ ,  $WW$ ,  $XX$ ,  $YY$ , and their partials) for each phase value until all the STM's are tested for stability.

## 7.2 Asynchronous Design

The asynchronous design method is exactly the same as the method for synchronous systems with random phase. The only difference is that the designer may want to specify higher process noise or measurement noise for the true asynchronous case.

The rationale for using the random-synchronous method for asynchronous sampling follows. Judicious selection of the asynchronous BTP will produce consecutive BTP STM's which are nearly equal. Small STM differences are simulated by additional process noise.

The recommended design procedure is to start with a cost function based on a successful continuous design method such as a good LQG design with a continuous controller. Then use the synchronous method to quickly refine the discrete gains at some arbitrary phase. Next, use the random-synchronous method to optimize the gains for random phasing. Finally, the asynchronous stability condition should be used to evaluate the stability of the resulting design. If the design is found wanting (because  $\bar{\sigma}(\Delta)$  is too large), then the process noise in the asynchronous states should be increased and the random-synchronous method repeated. The last gains are usually a good starting point for the next step.



# Chapter 8

## Design Examples

This chapter illustrates controller design using the methods developed in this report.

The first series examines the double integrator system of Figure 5.1. The design procedure is applied to the synchronous and asynchronous examples of Chapter 5. Step response plots and stability plots ( $\sigma$  and  $\lambda$  vs.  $\tau$ ) are presented for each set of resulting gains. Finally, the asynchronous sample rate is varied to show how the cost function can be used to evaluate sample rate effects.

The second series examines the two-link robot arm of [Ber86]. The synchronous design procedure is used to illustrate that this method duplicates the original result [Ber86]. Then, a new asynchronous sampling case is considered.

### 8.1 Double Integrator Examples

The states are numbered as shown in Figure 8.1. The parameters for all double integrator examples are summarized in Table 8.1. Cost weights and process noise matrices were adjusted by trial and error until the resulting design produced a reasonable step response. The cost weighting and noise covariance matrices were:

$$W_0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.1 & 0 \\ 0 & 0 & 0 & 0.1 \end{bmatrix} \text{ and } X^0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.1 & 0 \\ 0 & 0 & 0 & 0.1 \end{bmatrix}.$$

PARAMETER	Nominal	Optimal Synchronous	Optimal Asynchronous
C1	0.500	0.666	1.121
C2	0.500	0.115	0.429
C3	2.000	1.025	1.628
T1	1.0	1.0	1.0
T2	1.0	1.0	0.9

Table 8.1: Parameters for Double Integrator Examples

The same  $W_0$  and  $X^0$  matrices were used for all double integrator designs.

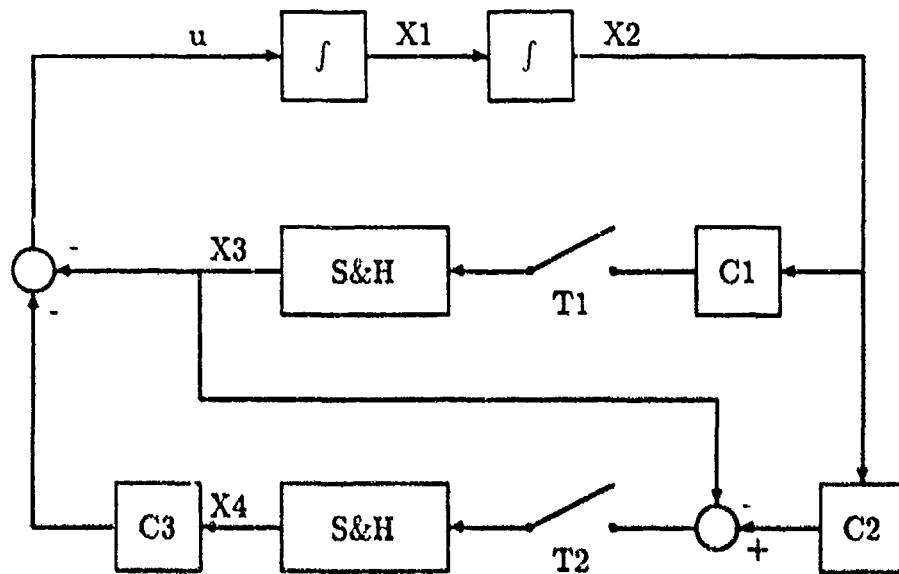


Figure 8.1: Block Diagram for Double Integrator System

### 8.1.1 Nominal Case

The nominal case used the gains from Chapter 5:  $c1=c2=0.5$  and  $c3=2$ . These values were picked at random. The stability plots for these cases were presented in Figures 4.1 and 4.2 of Chapter 4. The step responses are shown below in Figures 8.2

and 8.3. For the synchronous sampling case, the phase was 1.0.

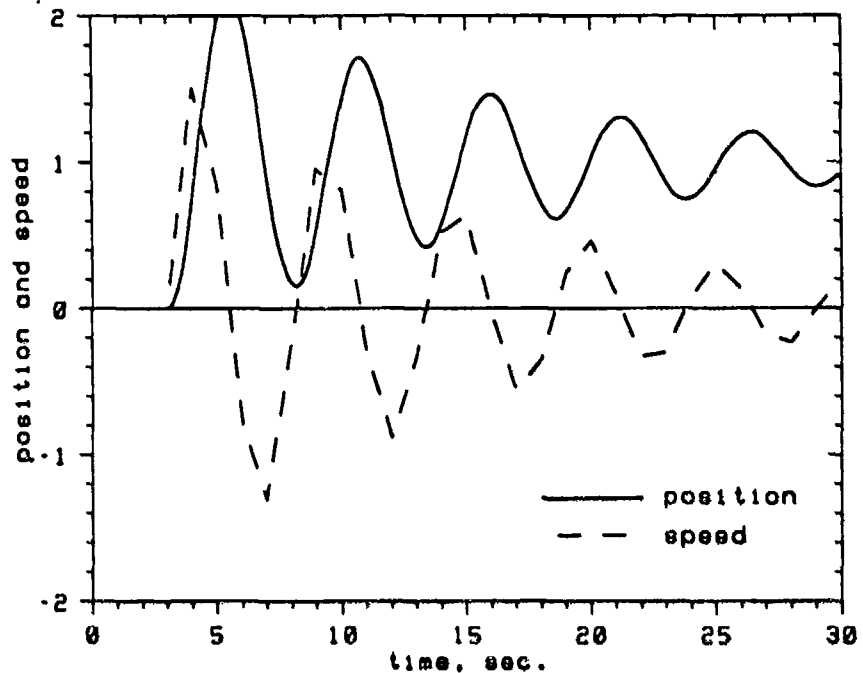


Figure 8.2: Step Response: Nominal Gains, Synchronous Sampling

Despite good stability (see Figure 4.1), the step response for the synchronous sampling case is poorly damped. Good stability as indicated by the  $\sigma$ - $\lambda$  plot should not be confused with good performance.

The step response with asynchronous sampling (Figure 8.3) shows the time-varying nature of the response. The damping and frequency vary (with period=9). This is typical time-varying behavior of strongly-coupled asynchronous systems.

### 8.1.2 Synchronous Case

The design procedure was used for the synchronous system at phase:  $\tau = 1$ . The optimal gains for this condition were:  $c_1=0.666$ ,  $c_2=0.115$  and  $c_3=1.025$ . The stability plot for these gains with synchronous sampling is shown in Figure 8.4. The system is guaranteed stable for all phase conditions except for a small range near zero ( $0 < \tau < 0.08$ ). The synchronous step response for phase=1.0 is shown in

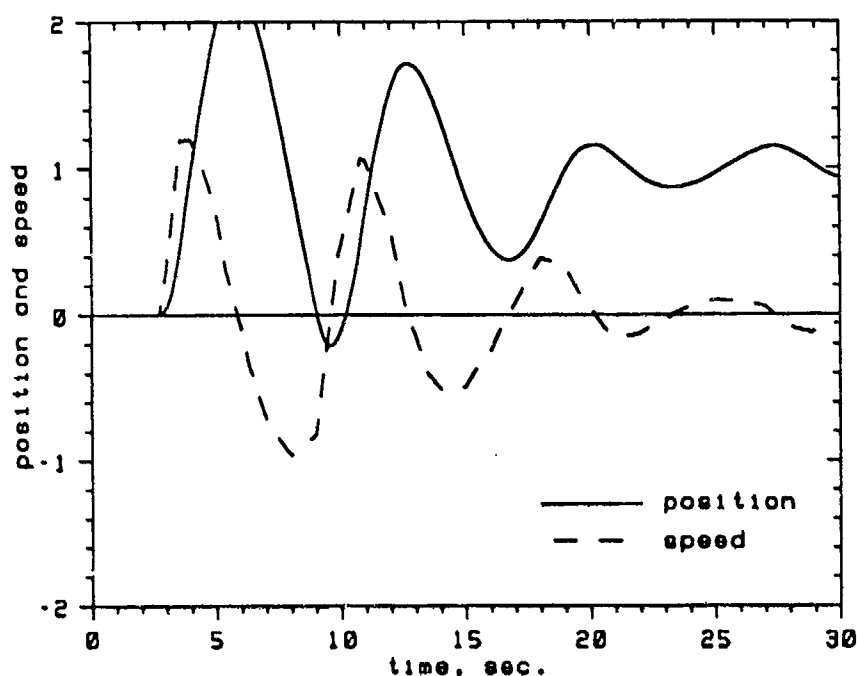


Figure 8.3: Step Response: Nominal Gains, Asynchronous Sampling

Figure 8.5. Overall, the stability and performance are good.

Using these same gains, these results were repeated for asynchronous sampling. The stability plot for these gains with asynchronous sampling is shown in Figure 8.6. The computed  $\sigma^*$  was 0.056 indicating that the system might be unstable (although it wasn't). The step response with asynchronous sampling is shown in Figure 8.7. The stability and performance deteriorated compared to the optimal (synchronous) design point.

### 8.1.3 Asynchronous Case

The design procedure was used for the asynchronous system. The optimal gains with asynchronous sampling were:  $c_1=1.121$ ,  $c_2=0.499$  and  $c_3=1.628$ . The stability plot for these gains with asynchronous sampling is shown in Figure 8.8. The computed  $\sigma^* = 0.061$  so the system is not guaranteed to be stable. Therefore, the stability test was repeated using a better approximation of synchronous sampling (the exact

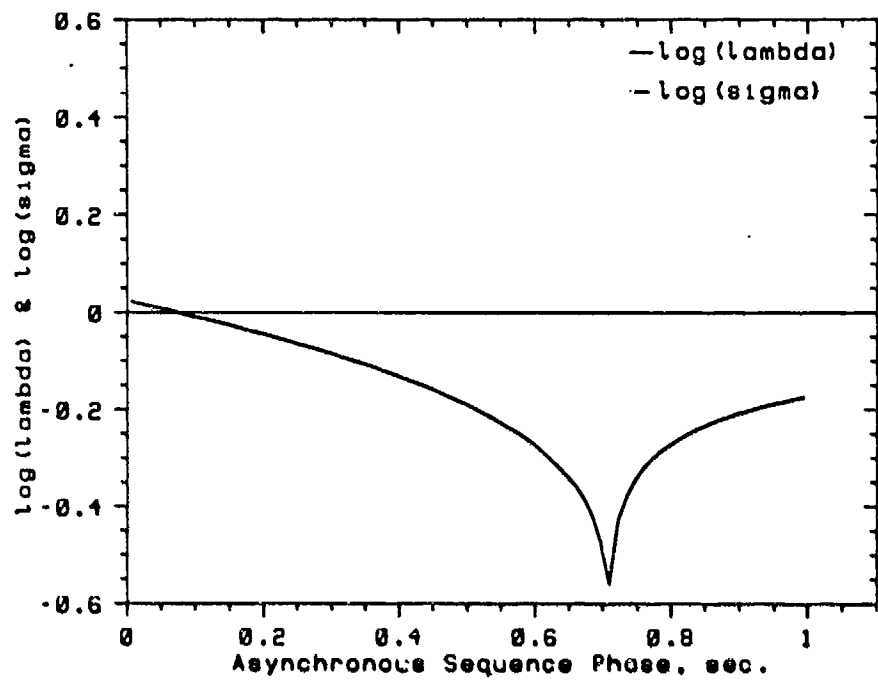


Figure 8.4: Synchronous Stability Plot, Synchronous Gains

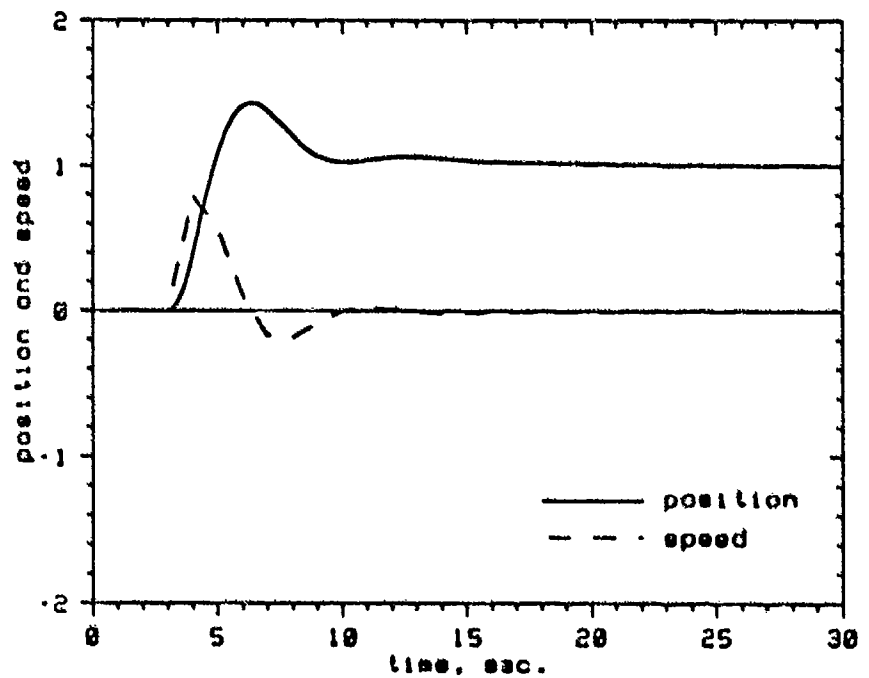


Figure 8.5: Step Response: Synchronous Gains, Synchronous Sampling

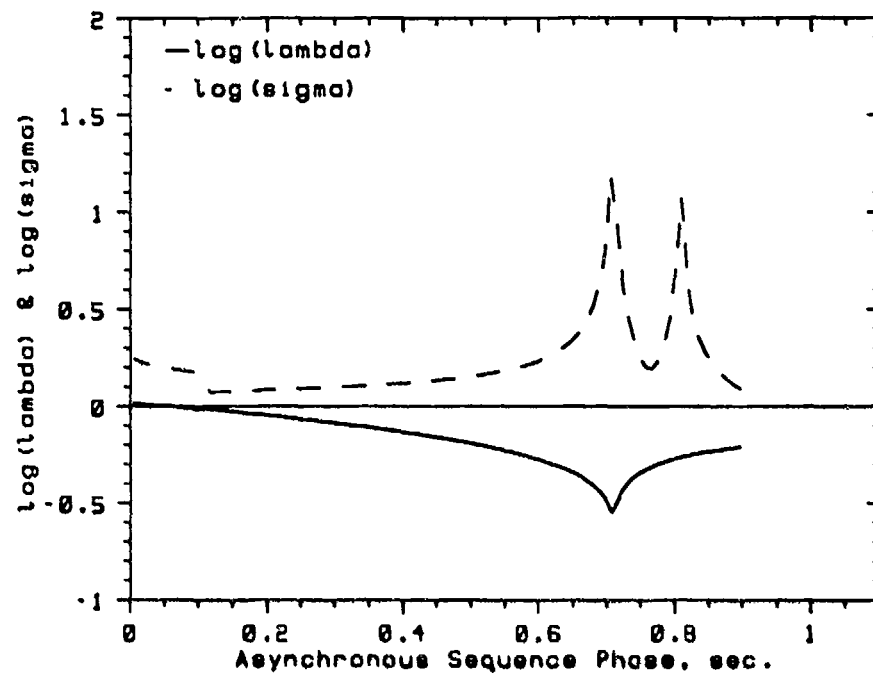


Figure 8.6: Asynchronous Stability Plot, Synchronous Gains

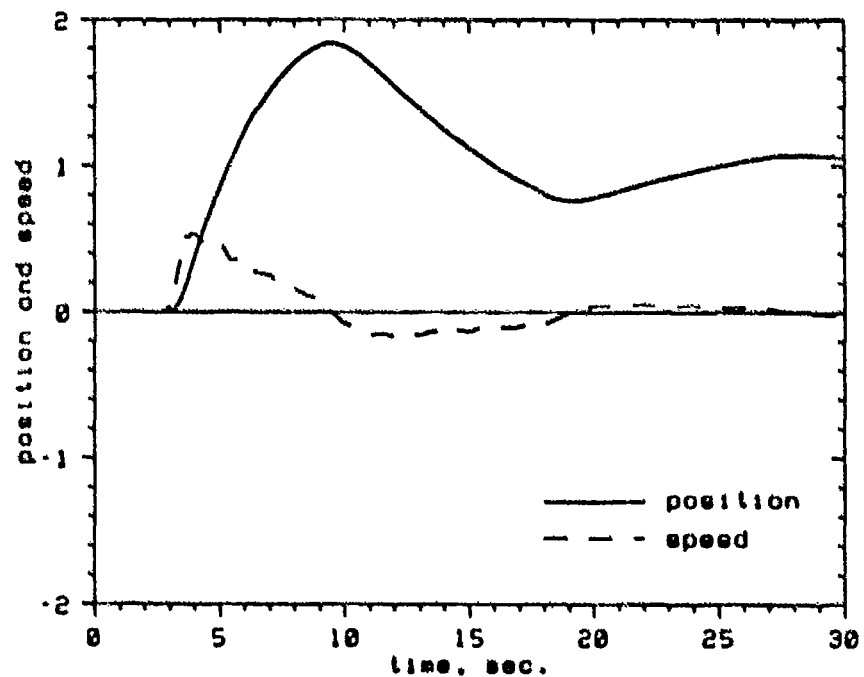


Figure 8.7: Step Response: Synchronous Gains, Asynchronous Sampling

BTP=9.0 was used). The resulting stability plot for the same gains with BTP=9.0 is shown in Figure 8.9. The asynchronous step response is shown in Figure 8.10. Overall, the stability and performance were good but not as good as the synchronous case.

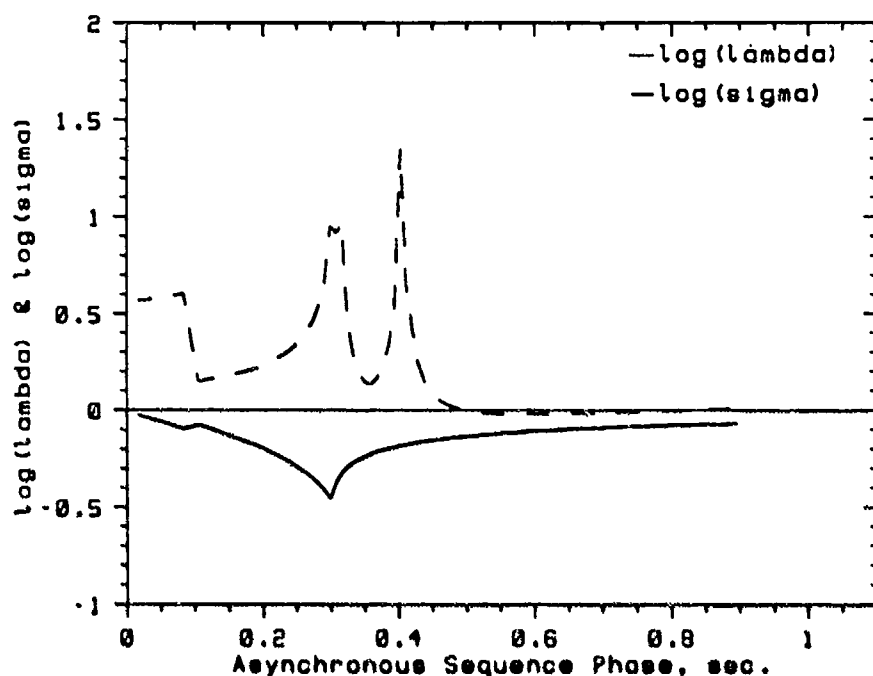


Figure 8.8: Asynchronous Stability Plot, Asynchronous Gains

Using these same gains, these results were repeated for synchronous sampling. The stability plot for these gains with synchronous sampling is shown in Figure 8.11. The synchronous step response for phase=1.0 is shown in Figure 8.12. The stability and performance deteriorated compared to the optimal asynchronous design point.

#### 8.1.4 Sample Rate Effects

To investigate the effect of sample rate selection, the period of the asynchronous (rate feedback) sampler was varied from 0.1 to 1.01 and an optimal asynchronous design was performed for each sample rate. The minimum cost as a function of sample period is shown in Figure 8.13.

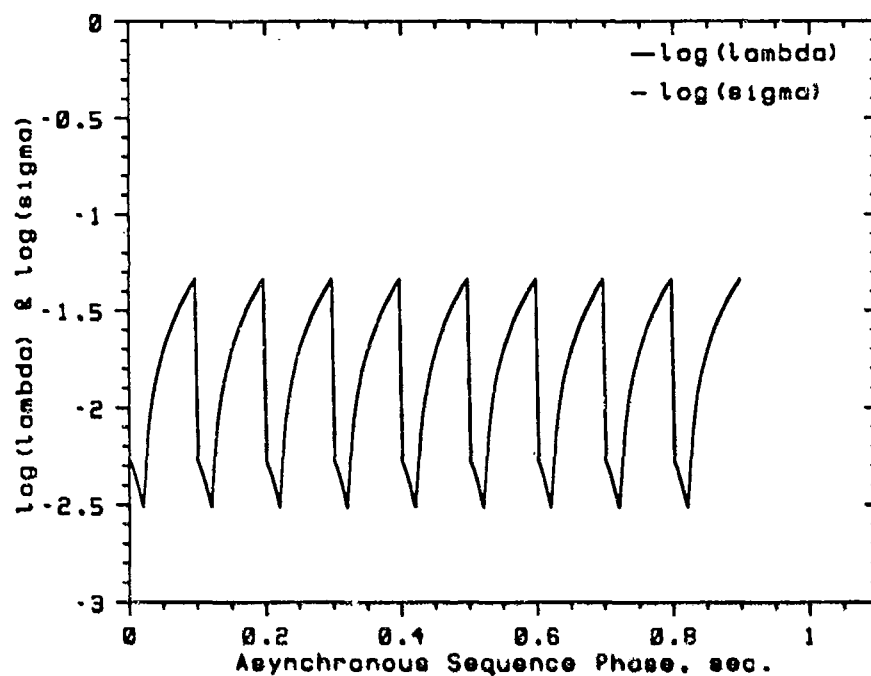


Figure 8.9: Asynchronous Stability Plot, Asynchronous Gains, Long BTP

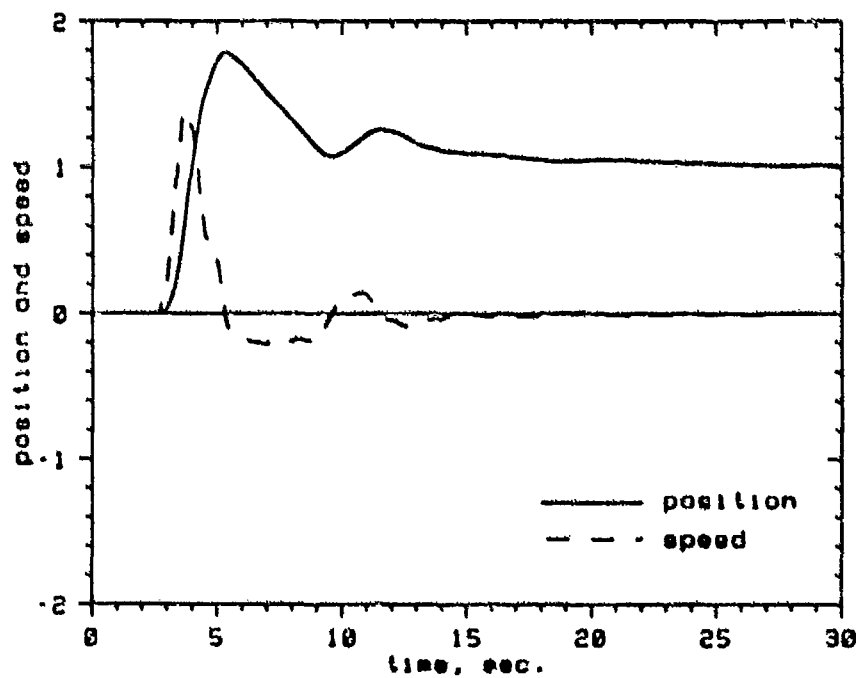


Figure 8.10: Step Response: Asynchronous Gains, Asynchronous Sampling



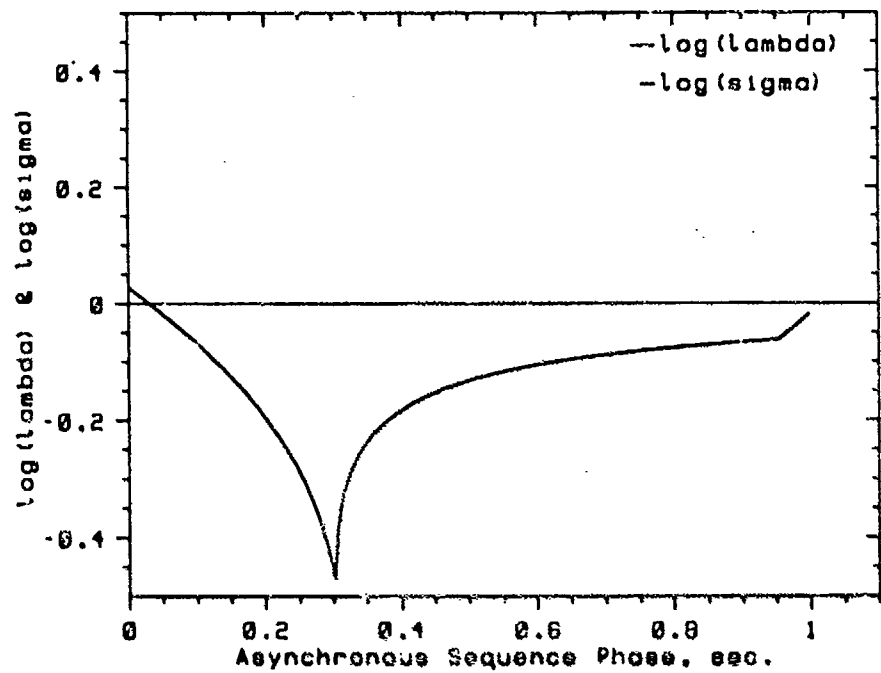


Figure 8.11: Synchronous Stability Plot, Asynchronous Gains

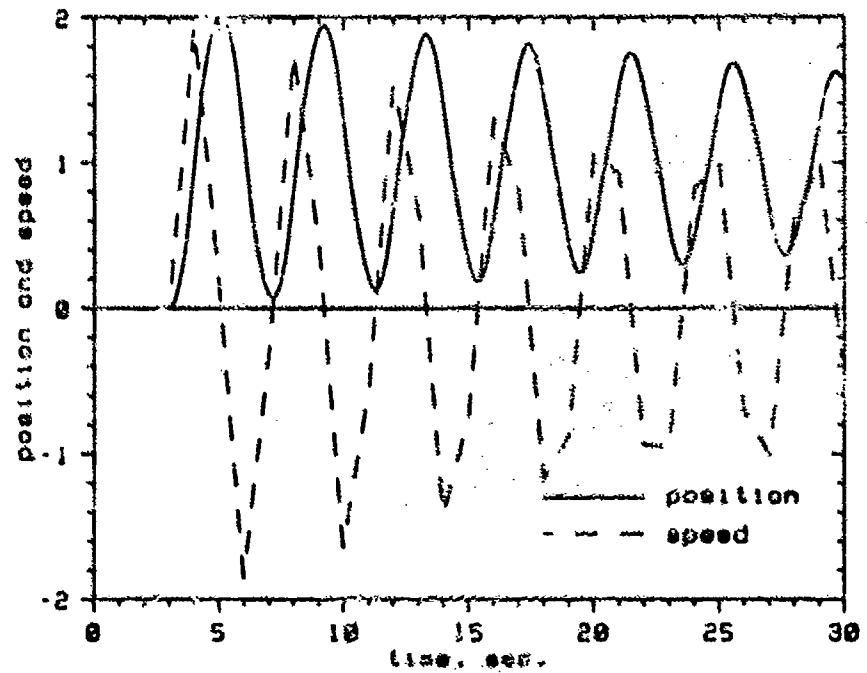


Figure 8.12: Step Response: Asynchronous Gains, Synchronous Sampling

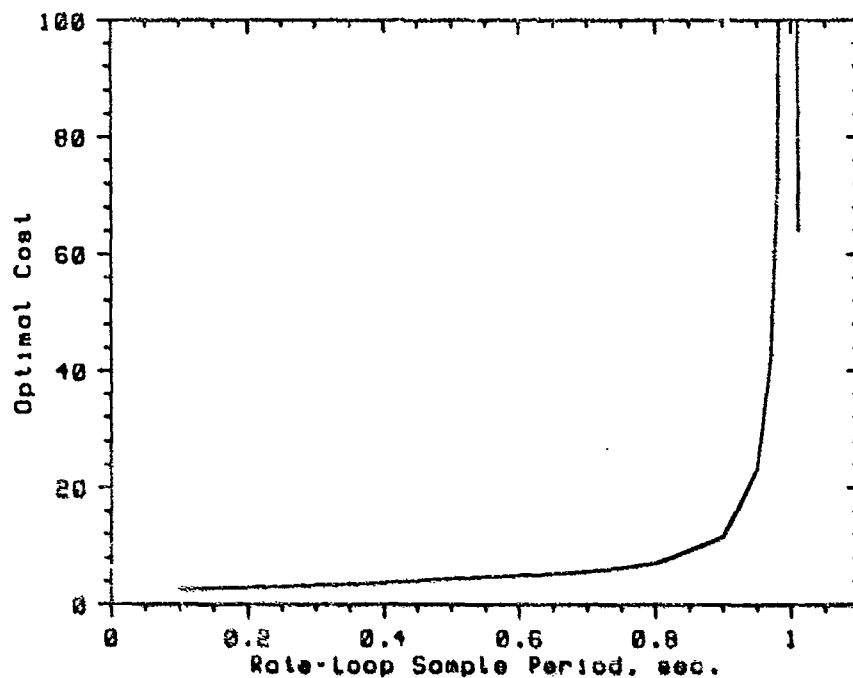


Figure 8.13: Optimal Cost as a Function of Sample Rate

Minimizing the cost function produced good controller designs. Therefore, we might use these minimum costs to compare the goodness of different sample rates. The figure illustrates the diminishing returns when one sampler becomes much faster than the other. The figure also illustrates an apparent worst case when both samplers operate at almost the same rate. At first glance, this seems to be a contradiction. For the earlier design case (see Figure 8.4) the cost was only about 3.5. Yet, Figure 8.13 shows the cost approaching 600 for the synchronous case.

The contradiction is easily solved. The optimal synchronous design case was based on a phase ( $\tau$ ) of 1.0. For the optimal synchronous (optimal at  $\tau = 1.0$ ) gains, Figure 8.4 shows that the plant is actually unstable at other values of  $\tau$ . The cost of an unstable system is infinite. Therefore, when the asynchronous design method is applied with sample rates approaching synchronous, the design at most phase conditions was compromised to achieve stability for the small, previously unstable, range of phase conditions. So the contradiction is solved. The design method gives the conservative result: a design that is stable (although just barely).

at all phase conditions.

## 8.2 Two-Link Arm Examples

A more realistic example is used now. The two-link robot arm and controller of [Ber86] is used. This system is shown in Figure 8.14. Two sampling configurations were examined. For the synchronous configuration, the slow samplers ( $T_1$ ) operated with a period of 0.225 seconds and the fast samplers ( $T_2$ ) operated eight times faster. For the asynchronous configuration, the slow samplers operated at the same rate but the fast samplers operated at  $2\pi$  times that rate. This was truly asynchronous (to the computation accuracy of the machine). The cost weighting and noise covariance matrices were identical to those in [Ber86]:

$$W_0 = \begin{bmatrix} 21 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1850 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 69.44 & 69.44 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 69.44 & 69.44 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

and

$$X^0 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 915.48 & 0 & -2976.4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -2976.4 & 0 & 14874.1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

The same  $W_0$  and  $X^0$  matrices were used for both designs.

The parameters for the two-link arm examples are summarized in Table 8.2.

PARAMETER	Optimal Synchronous	Optimal Asynchronous
$\alpha_1$	-0.485	-0.455
$\beta_{11}$	11.297	11.046
$\beta_{12}$	0.393	0.691
$\gamma_{11}$	-13.483	-12.888
$\gamma_{12}$	1.071	0.568
$\alpha_2$	-0.553	-0.543
$\beta_{21}$	0.098	0.095
$\beta_{22}$	13.439	9.597
$\gamma_{21}$	-0.121	-0.115
$\gamma_{22}$	-16.865	-11.710
T1	0.225	0.225
T2	T1/8	T1/(2 $\pi$ )

Table 8.2: Parameters for Two-Link Arm Examples

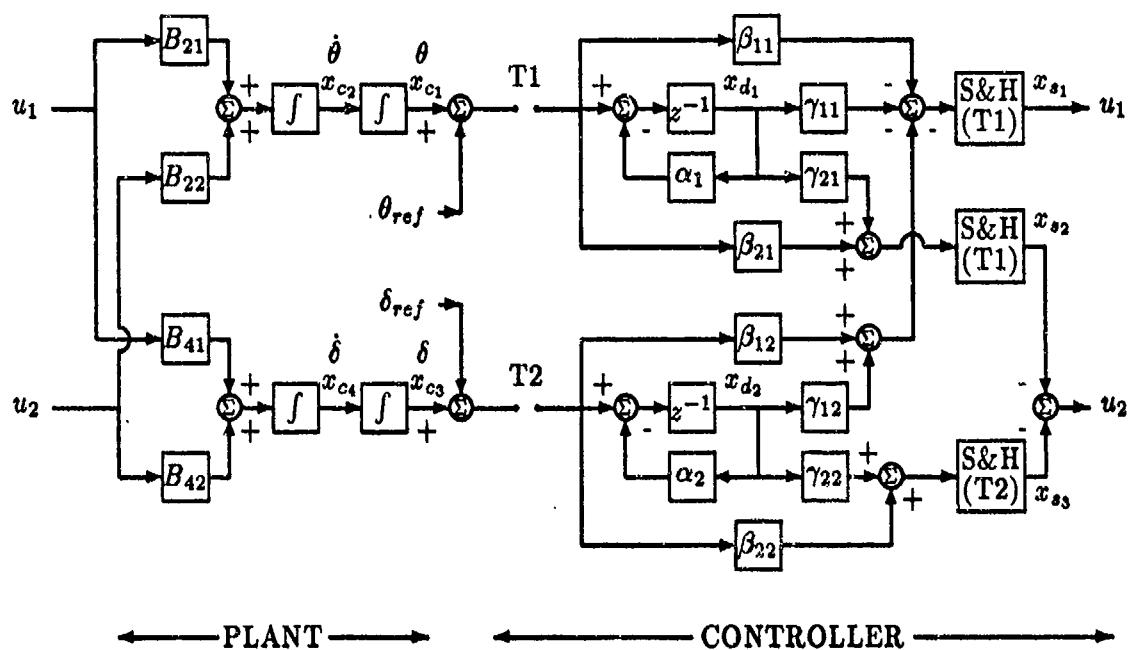


Figure 8.14: Block Diagram for Two-Link Robot

The two-link arm system has a fairly high sample rate and it does not have strong coupling between controllers. The resulting asynchronous system performs well at all gain/sampling combinations, but the best performance occurred when the design gains are used for each sample condition.

### 8.2.1 Nominal Case

The new synchronous constrained optimization algorithm was applied to the multi-rate two-link arm system of [Ber86] and the resulting gains were identical to those published in [Ber86].

The asynchronous stability plot for those gains is shown in Figure 8.15. This plot shows that the stability is totally insensitive to sample sequence phase. The corresponding step response is shown in Figure 8.16.

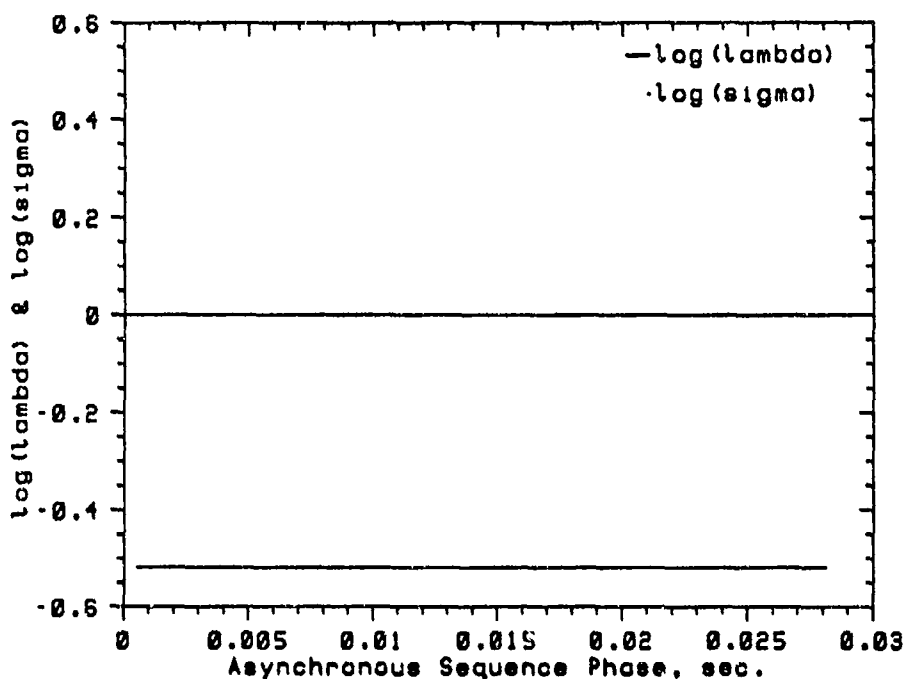


Figure 8.15: Synchronous Stability Plot, Synchronous Gains

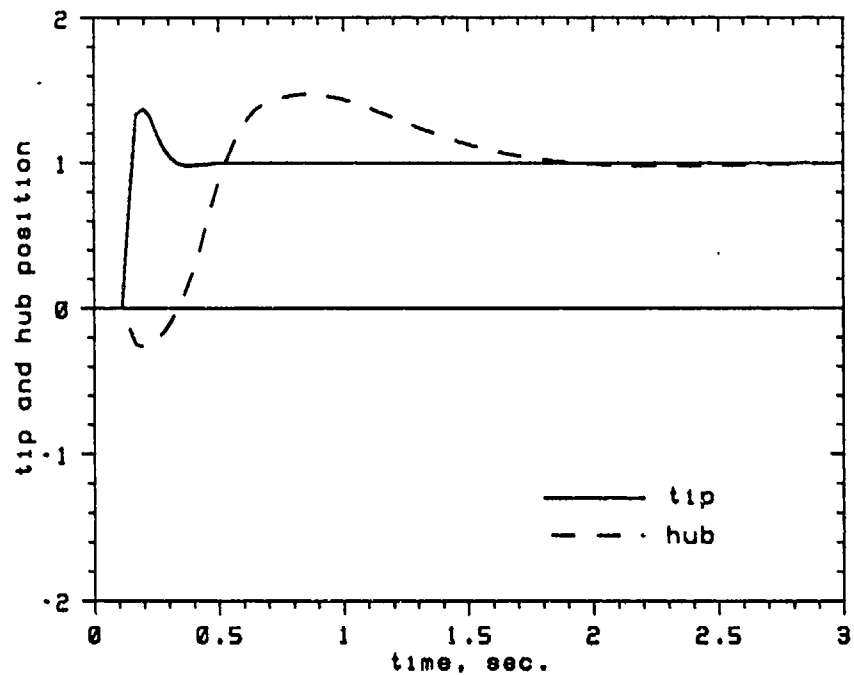


Figure 8.16: Step Response: Synchronous Gains, Synchronous Sampling

### 8.2.2 Asynchronous Case

To evaluate asynchronous sampling, the step response was repeated using the same gains but with the asynchronous sampling condition. The result is shown Figure 8.17. The step response is somewhat degraded.

Some step response degradation resulted from the overall slower sampling rate and from the mismatch between gains and sample rates. To investigate this, the asynchronous design algorithm was used to find a new set of optimal gains for the asynchronous sampling condition. The resulting gains are shown in Table 8.2. The asynchronous stability plot for the new gains is shown in Figure 8.18. The step response with the asynchronous gains and asynchronous sampling is shown on Figure 8.19. Except for a slight difference in initial overshoot, this response is nearly identical to the optimal synchronous response.

Finally, the step response was repeated for the asynchronous gains and the synchronous sampling condition (see Figure 8.20). This response is slightly slower

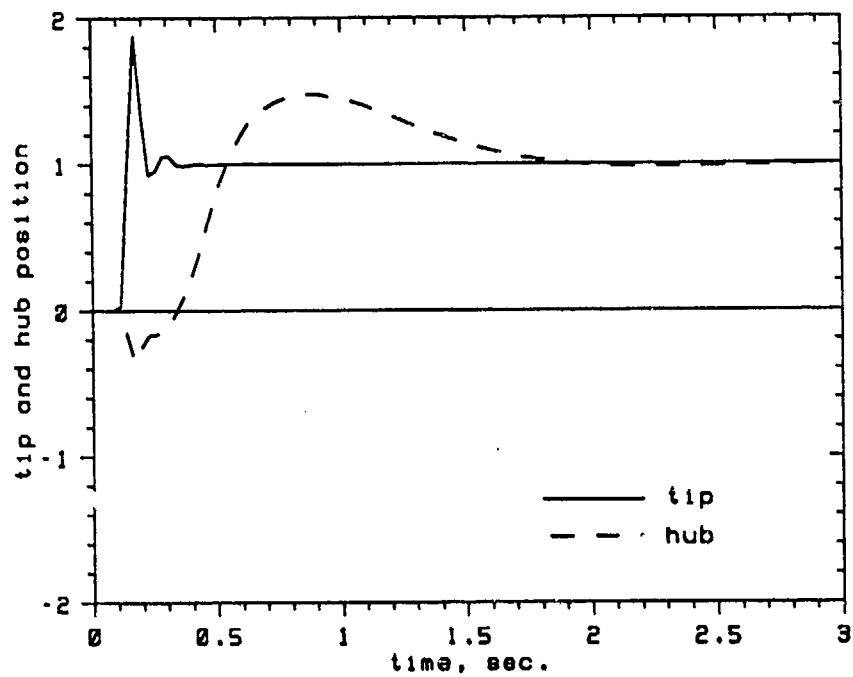


Figure 8.17: Step Response: Synchronous Gains, Asynchronous Sampling

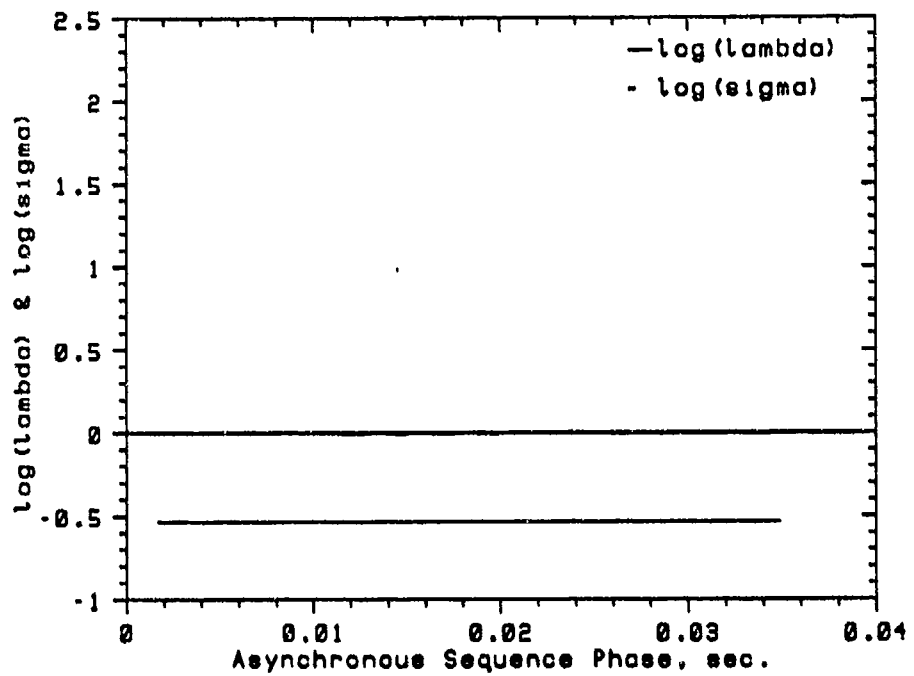


Figure 8.18: Asynchronous Stability Plot, Asynchronous Gains



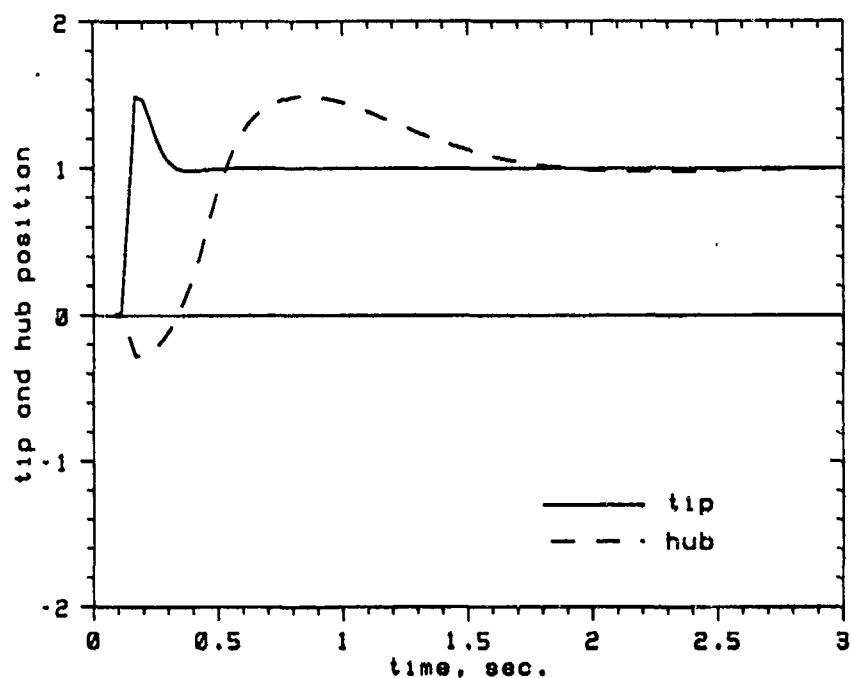


Figure 8.19: Step Response: Asynchronous Gains, Asynchronous Sampling

than the best results but quite good overall.

For the two-link arm system, the asynchronous controller is nearly as good as the best synchronous controller.

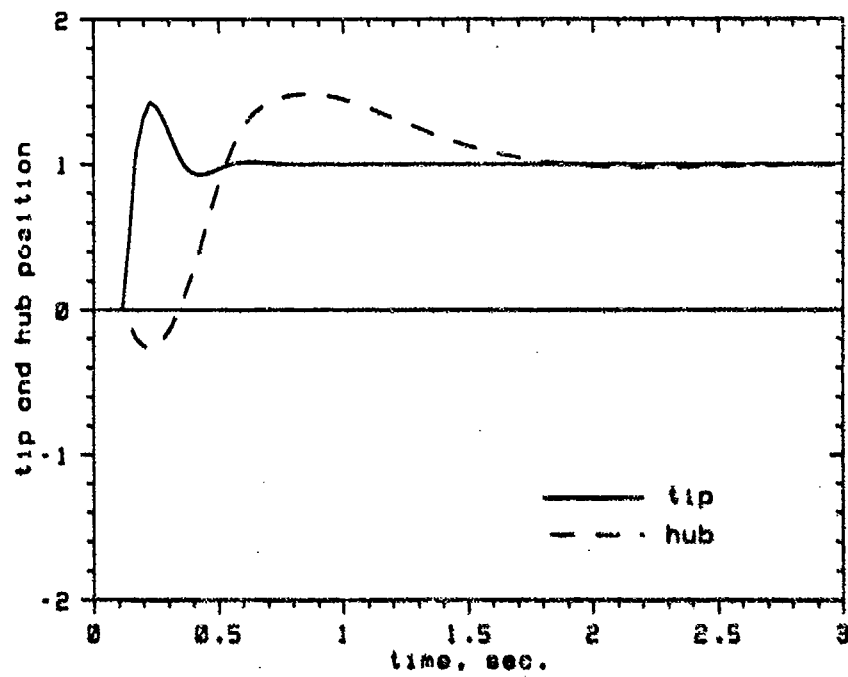


Figure 8.20: Step Response: Asynchronous Gains, Synchronous Sampling

## Chapter 9

# Practical Considerations

### 9.1 BTP Selection

For the asynchronous case, BTP selection is left to the designer. There are an infinite number of BTP choices. Long BTP's provide better approximations of synchronous sampling but long BTP's require more calculation (for  $\Phi$ ,  $WW$ ,  $XX$ ,  $YY$ , and their partials). Furthermore, even slight improvements in the BTP match eventually require large increases in the BTP. Clearly, this is a diminishing returns situation. Incidentally, the design method is not compromised by long BTP's (although the computational burden is high) because the cost is the true continuous-time integral of the weighted mean square error. However, the sufficient stability condition only considers errors at the ends of BTP's; so short BTP's may be more appropriate for the stability test.

The following rules and rationale were developed to guide the best selection from a finite number of alternatives. Suppose now, that the designer starts with a list of candidate BTP's. For example, this may be all candidates with BTP's shorter than one minute. Such a choice would be reasonable if the plant time constants were on the order of a minute.

### 9.1.1 Rule of Thumb

Let  $T_1, T_2, \dots, T_k$  be candidate BTP's which contain  $n_1, n_2, \dots, n_k$  discrete events and which have consecutive phase slips  $\zeta_1, \zeta_2, \dots, \zeta_k$ . The best BTP is the one with the lowest  $n_j * \zeta_j$  product.

Consecutive phase slip is the difference in phase between consecutive BTP's. Phase is defined as the delay from the start of the BTP to the start of the asynchronous sample sequence. The other sequence is called the synchronous sequence because it is synchronous with the BTP.

### 9.1.2 Rationale

Recall that the continuous-time state transition matrix for elapsed time  $\zeta$  can be computed as:

$$\Phi(\zeta) = \begin{bmatrix} e^{\begin{bmatrix} A & B \\ 0 & 0 \end{bmatrix} \zeta} & 0 \\ 0 & I \end{bmatrix}.$$

Now assume that  $\zeta$  is small so this can be approximated by the first two terms of the series expansion for the exponential:

$$\Phi(\zeta) \approx I + \begin{bmatrix} A & B & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \zeta = I + Z\zeta.$$

where  $Z$  is defined as indicated.

Let sequence 1 be synchronous with the BTP while the sequence 2 phasing "slips" by  $\zeta$  from one BTP to the next. Suppose we expand the state transition matrix for the first BTP as:

$$\Psi_1 = D_1 \Pi_1 D_2 \Pi_2 \cdots D_{n-1} \Pi_{n-1} D_n \Pi_n$$

where  $D_i$  is a discrete transition in sequence 1 and  $\Pi_i$  includes all the continuous transitions and all the discrete transitions from sequence 2. Then the state transition matrix for the next BTP will be

$$\Psi_2 = D_1 \Phi(\zeta) \Pi_1 \Phi(-\zeta) D_2 \Phi(\zeta) \Pi_2 \cdots \Pi_{n-1} \Phi(-\zeta) D_n \Phi(\zeta) \Pi_n \Phi(-\zeta)$$

providing that none of the sequence 1 discrete transitions transpose with sequence 2 discrete transitions. Using these approximations for  $\Phi$  we get:

$$\begin{aligned}\Psi_2 &\approx D_1(I + Z\zeta)\Pi_1(I - Z\zeta)D_2(I + Z\zeta)\Pi_2\cdots\Pi_{n-1}(I - Z\zeta)D_n(I + Z\zeta)\Pi_n(I - Z\zeta) \\ &= \Psi_1 + \zeta[ \begin{aligned} &D_1Z\Pi_1D_2\Pi_2\cdots D_{n-1}\Pi_{n-1}D_n\Pi_n \\ &- D_1\Pi_1ZD_2\Pi_2\cdots D_{n-1}\Pi_{n-1}D_n\Pi_n \\ &\vdots \\ &D_1\Pi_1D_2\Pi_2\cdots D_{n-1}\Pi_{n-1}D_nZ\Pi_n \\ &- D_1\Pi_1D_2\Pi_2\cdots D_{n-1}\Pi_{n-1}D_nZ\Pi_n \end{aligned} ] \end{aligned} \quad (9.1)$$

The basis of the asynchronous analysis and design approaches is successive BTP state transition matrices which are nearly equal. This means that the last term in Equation 9.1 should be small. We cannot say much about this term, but two things are clear. First  $\zeta$  multiplies the whole thing so  $\zeta$  should be small. Second, the number of terms in the brackets is proportional to the number of discrete events in schedule 1 during the BTP. So the number of discrete synchronous events during the BTP should be small. The rule of thumb follows from minimizing the product of these two items.

## 9.2 Asynchronous Analysis of Synchronous Systems

It may be more appropriate to design and analyze a synchronous system as if it were asynchronous. For example, assume a system is synchronous but the real BTP is very long (much longer than plant time constants).

The computations would be much easier if the system were treated as asynchronous with a shorter BTP. Furthermore, we are normally interested in behavior during time intervals much shorter than the BTP. Although state errors may decrease over the whole (long) BTP, they may grow unacceptably during some portions of the BTP (i.e. short term instability). When behavior during periods shorter than the BTP is of interest, it is appropriate to use a short "fake" BTP and analyze the

system as if it were asynchronous. In this case,  $\tau$  is not uniformly distributed; rather, it assumes a finite number of values depending on the initial phasing and sample rate. Hence, only a few evenly-spaced points on the stability curves are relevant. Still,  $\sigma^*$  and the  $\lambda$  and  $\sigma$  curves give good estimates of the stability, particularly if the exact phasing is unknown.

### 9.3 Accelerated Convergence

The design method is completely automatic; however, some manual interaction can accelerate convergence to the optimal solution. The synchronous optimization process is much faster (at least three times faster) than asynchronous optimization. Consequently, the synchronous method should be used with some representative phase condition to find initial gains for the asynchronous optimization process.

The asynchronous optimization search cannot switch to the stable cost function until stabilizing gains are found for all of the phase conditions (curve-fit points). When only a small phase region is unstable, this search can be very slow or it can stall altogether. This is usually cured by one or two synchronous optimization steps at the offending phase condition. Alternatively, if there are several small disjoint unstable regions, the power in the initial cost function can be increased. Fourth and eighth powers were used in the examples but these can be increased within the numerical precision of the computing hardware.

### 9.4 Cost Weighting Matrices

Choosing good noise covariance and cost weighting matrices is not simple. [Ber86] demonstrated that matrices selected to produce good continuous-time designs also worked well for the multirate controllers. Based on this and the relative ease of continuous-time design methods, these matrices may be developed by first designing a "good" continuous-time LQG controller and using the same cost and covariance matrices for the multirate discrete design.

## 9.5 Numerical Properties

The design and analysis calculations rely on two matrix operations that may present numerical problems. These are spectral factorization and matrix exponentiation. While these two matrix functions always exist (in theory) computing their values can be a significant numerical problem. The codes for the sample problems used PC-MATLAB [MLBK85] which implements the EISPACK algorithm [SBD\*76] and the Padé approximation algorithm [MVL78]. One attempt to use another matrix package (which used an eigensystem approach for matrix exponentiation) was unreliable. Any implementation of the methods presented in this paper should start with fast reliable algorithms for these two matrix functions.

### Maximum System Size

Practically, the maximum system size for the method is limited by the numerical precision of the various mathematical operations. Assuming that the BTP is reasonably short, the various STM-related matrices ( $\Phi$ ,  $WW$ ,  $XX$ ,  $YY$ , and their partials) can be computed with only small errors since matrix multiplication and addition are the only operations involved. Also, the closed-loop system eigenvector matrix (complex) must be inverted, but this should be a well-conditioned matrix in the vicinity of the optimal gains (otherwise,  $S^{-1}XXS^{-H}$  in equation 6.9 gives high costs). Other inversion processes use the pseudo-inverse so ill conditioning is not a problem.

With no analog gains (in  $A$  or  $B$ ), finding the  $Q$  matrix requires exponentiating a  $2(n_c + n_s)$  by  $2(n_c + n_s)$  matrix where  $n_c$  is the number of continuous states and  $n_s$  is the number of sample states. If there are variable gains in the  $A$  or  $B$  matrices, a  $3(n_c + n_s)$  by  $3(n_c + n_s)$  matrix must be exponentiated to find  $Q$ . Also, the STM eigensystem (dimension equals number of states) must be found for each gain/phase condition evaluated. The ability to accurately find these transcendental matrix functions limits the size of the system these methods can handle.

No attempt was made to quantitatively determine the maximum system size as a function of algorithm accuracy. The PC-MATLAB routines use the IEEE

standard arithmetic in the 8087 numeric coprocessor. Each real value is 64 bits (52 for the mantissa). Unusual behavior attributable to numeric problems was not encountered.

## 9.6 PC-MATLAB Implementation

PC-MATLAB was an excellent vehicle for developing the prototype algorithm. The compile and execute speed were good, but more importantly, the high-level MATLAB language allowed very rapid, reliable code development. Furthermore, the code is quite readable. MATLAB's modularity and consistency checks assisted code debugging significantly. However these very strengths created limitations that could be significant if execution speed and storage efficiency are paramount considerations.

### Pointers and Tables

PC-MATLAB executes built-in functions (like matrix exponentiation and modal decomposition) very quickly but logical constructs ("for" loops or "if" loops) execute slowly. Therefore, brute-force calculations were often used in place of seemingly more efficient constructs. For example, in PC-MATLAB, it was much faster to recompute a frequently-used matrix exponential than it was to branch to a subroutine for table look-up. Similarly, it was faster to store a sparse matrix and multiply by the whole matrix than to use pointers to the few non-zero elements. For implementation in some other compiled language, significant speed and storage improvements may be obtained through the use of pointers and table look-ups.

### 9.6.1 Square Root Algorithms

Several symmetric matrices ( $WW$ ,  $XX$ , and the Hessian) are propagated using updates of the form:

$$Z_{n+1} = T Z_n T^T + Y,$$

where  $Y$  and  $Z$  are symmetric and non-negative definite. This is the same form as the Kalman Filter covariance propagation equation. That suggests using a square



root algorithm [Kai81] [BH75] to accomplish the propagation. This was not attempted in the prototype code because MATLAB only deals with rectangular matrices and "definiteness" problems were not encountered. Therefore, there was no storage or computational advantage to implementing the square root algorithm in the PC-MATLAB code. However, implementations in compiled languages that can exploit the storage and computational advantages of triangular matrices would benefit from a square root propagation algorithm.



# Chapter 10

## Summary and Recommendations

### 10.1 Summary

A sufficient asynchronous stability condition was developed. The figure of merit describes an exponential envelope which bounds the worst-case average state error history for any initial condition. The figure of merit is found through numerical integration of quantities computed from the system state transition matrix. The integrands have useful physical interpretations.

The Constrained Optimization Method of Berg [Ber86] was generalized and reformulated. The resulting method is more efficient, it avoids numerical overflow problems, and it includes discrete measurement noise. The reformulated method was shown to replicate Berg's results.

The reformulated constrained optimization method was extended to the synchronous sampling case with random phasing. This extended approach produced satisfactory controllers for several asynchronous sampling cases.

### 10.2 Recommendations for Future Research

Recoding the method in a compiled language is tedious but could provide substantial speed and storage improvements. An improved linear search algorithm should give an immediate speed improvement. The other approaches listed in the last chapter

(square root algorithms and table look-ups) should also provide more speed.

Modal decomposition accuracy and matrix exponentiation accuracy appear to be the limiting factors on system size. This relationship could be explored to find the practical limits of the method.

The robustness with respect to sample rate, controller gain, and plant uncertainty should be investigated. Designs resulting from this method should have good robustness since they minimize a continuous LQR cost function and continuous LQR designs have desirable robustness properties. However, nothing is actually known about the robustness properties of these designs except that the cost gradient is zero (with respect to the gains).

Methods of selecting sample rates should be investigated. All existing design methods begin with specified sample rates. If some equivalence between controller computational operations and cost function is specified, then an optimal sample rate should exist.

Methods of selecting the cost function weights and process noise should be investigated. Existing design methods begin with specified values for these matrices ( $W_0$  and  $X^0$  in this paper). Berg [Ber86] suggested using a satisfactory continuous LQR design as the basis for the diagonal blocks and this seems to work well. However, use of the off-diagonal blocks to avoid saturation may merit further investigation.

Methods of balancing the eigenvector matrix should be investigated. The sufficient stability condition is sensitive to the scaling of the BTP STM ( $\Psi$ ) eigenvectors. This also affects the numerical precision of the design algorithm. The optimal scaling could be investigated.

Methods of extending the method to systems with defective and ill-conditioned BTP STM's should be investigated. The design method used the modal decomposition as a convenient way to solve the Liapunov equation for the steady-state covariance. However, the steady-state covariance exists for all stable systems. Consequently, the method should extend to all cases given another solution to the Liapunov equation.

# Appendix A

## Eigensystem Derivatives

This appendix develops formulas for the first partial derivatives of a matrix eigenvalue and eigenvector as a function of the matrix, its eigenvalue, its eigenvector, and the first partial derivative of the matrix.

### A.1 Problem Statement

Consider any non-defective real  $n$  by  $n$  matrix  $Z$ . Let  $\lambda_i$  be an isolated non-zero eigenvalue and  $s_i$  the corresponding eigenvector.

Since  $Z$  is not defective, it can be factored:

$$Z = S\Lambda S^{-1} \quad (\text{A.1})$$

where  $S = [s_1 \cdots s_i \cdots s_n]$  is a matrix of the  $n$  independent eigenvectors and  $\Lambda = \text{diag}[\lambda_1 \cdots \lambda_i \cdots \lambda_n]$  is a diagonal matrix of the corresponding eigenvalues in the same order.

Since  $\lambda_i$  and  $s_i$  are an eigen-pair:

$$Zs_i = \lambda_i s_i \quad (\text{A.2})$$

by definition. Also, the norm of  $s_i$  is arbitrary so choose  $\|s_i\|_2 = 1$  for convenience.

Suppose  $Z$  is actually  $Z(\alpha)$  where  $\alpha$  is some scalar. Define

$$\dot{Z} = \frac{\partial Z}{\partial \alpha}, \quad \dot{\lambda}_i = \frac{\partial \lambda_i}{\partial \alpha}, \quad \text{and} \quad \dot{s}_i = \frac{\partial s_i}{\partial \alpha}.$$

Then, the problem is to compute  $\dot{\lambda}_i$  and  $\dot{s}_i$  given  $Z$ ,  $\dot{Z}$ ,  $\lambda_i$ , and  $s_i$ .

Since the norm of  $s_i$  is arbitrary, let

$$s_i^H s_i = 1 \quad (\text{A.3})$$

where the "H" superscript indicates the conjugate transpose (Hermetian). Differentiating with respect to  $\alpha$  gives  $\dot{s}_i^H s_i + s_i^H \dot{s}_i = 0$  or

$$\Re(\dot{s}_i^H s_i) = \Re(s_i^H \dot{s}_i) = 0 \quad (\text{A.4})$$

This means that, for a constant-length eigenvector,  $\dot{s}_i$  can be expressed as  $\dot{s}_i = \dot{s}_i^\perp + j\gamma s_i$ , where  $\dot{s}_i^\perp$  is in the orthogonal complement subspace of  $s_i$  so  $s_i^H \dot{s}_i^\perp = 0$ ,  $j = \sqrt{-1}$ , and  $\gamma$  is some real constant.

## A.2 Equation Development

Start by differentiating Equation A.2 with respect to  $\alpha$ :

$$\dot{Z}s_i + Z\dot{s}_i = \dot{\lambda}_i s_i + \lambda_i \dot{s}_i. \quad (\text{A.5})$$

Observe that the right-hand side is already divided into a component in the  $s_i$  direction and a part in the subspace where  $\dot{s}_i$  resides. Pre-multiplying both sides by  $s_i^H$  and solving for  $\dot{\lambda}_i$ :

$$\dot{\lambda}_i = s_i^H \dot{Z}s_i + s_i^H (Z - \lambda_i I) \dot{s}_i.$$

Observe that  $s_i$  is in the null space of  $(Z - \lambda_i I)$  so if  $\dot{s}_i = \dot{s}_i^\perp + j\gamma s_i$ , only  $\dot{s}_i^\perp$  influences  $\dot{\lambda}_i$  and  $\gamma$  is irrelevant. Using this fact, we can simplify the equation somewhat:

$$\dot{\lambda}_i = s_i^H \dot{Z}s_i + s_i^H (Z - \lambda_i I) \dot{s}_i^\perp = s_i^H \dot{Z}s_i + s_i^H Z \dot{s}_i^\perp. \quad (\text{A.6})$$

Now, pre-multiply Equation A.5 by  $(I - s_i s_i^H)$  to project into the orthogonal complement subspace of  $s_i$ :

$$(I - s_i s_i^H) \dot{Z}s_i + (I - s_i s_i^H) Z \dot{s}_i = \lambda_i (I - s_i s_i^H) \dot{s}_i$$

which can be regrouped as

$$(I - s_i s_i^H) \dot{Z} s_i = (I - s_i s_i^H) [\lambda_i I - Z] (\dot{s}_i^\perp + j\gamma s_i) = [\lambda_i I - (I - s_i s_i^H) Z] \dot{s}_i^\perp. \quad (A.7)$$

The unique solution is given by

$$\dot{s}_i^\perp = [\lambda_i I - (I - s_i s_i^H) Z]^{-1} (I - s_i s_i^H) \dot{Z} s_i \quad (A.8)$$

if the matrix  $[\lambda_i I - (I - s_i s_i^H) Z]$  is not singular. This non-singularity requirement will be explored further in the last section.

If Equation A.8 has a unique solution, then it can be used to compute  $\dot{s}_i^\perp$  and that result can be used in Equation A.6 to compute the respective  $\dot{\lambda}_i$ . If  $\lambda_i$  and  $s_i$  are available, calculations require solving one set of complex linear simultaneous equations (of form  $Ax = b$ ), four complex vector-matrix multiplications and five complex vector products.

### A.2.1 Eigenvector Derivatives

The eigenvector derivatives may require an extra step because  $\dot{s}_i = \dot{s}_i^\perp + j\gamma s_i$  and  $\gamma$  is still unknown. For  $s_i$  real,  $\dot{S}_i$  will also be real so  $\gamma$  is zero. Hence,  $\dot{s}_i = \dot{s}_i^\perp$  for real eigenvectors.

For complex eigenvectors, the solution requires separate consideration of the real or imaginary parts. The real part of Equation A.5 can be written:

$$\dot{Z}v + Z(\Re(\dot{s}_i^\perp) - \gamma w) = \dot{\sigma}v - \dot{\omega}w + \sigma(\Re(\dot{s}_i^\perp) - \gamma w) + \omega(\Im(\dot{s}_i^\perp) + \gamma v).$$

where  $s_i = v + jw$  and  $\lambda = \sigma + j\omega$ . But  $\gamma$  is the only unknown ( $v$ ,  $w$ ,  $\sigma$ ,  $\dot{\sigma}$ ,  $\omega$ ,  $\dot{\omega}$ , and  $\dot{s}_i^\perp$  are known). Therefore

$$[\dot{Z} - \dot{\sigma}I]v + \dot{\omega}w + [Z - \sigma I]\Re(\dot{s}_i^\perp) - \omega\Im(\dot{s}_i^\perp) = \gamma([Z - \sigma I]w + \omega v) \quad (A.9)$$

where everything is known except the real scalar  $\gamma$ . Any non-zero component is sufficient to compute  $\gamma$ . In practice, the vector  $([Z - \sigma I]w + \omega v)$  should be computed and the largest component should be used to find  $\gamma$ . Then only the corresponding component of  $[\dot{Z} - \dot{\sigma}I]v + \dot{\omega}w + [Z - \sigma I]\Re(\dot{s}_i^\perp) - \omega\Im(\dot{s}_i^\perp)$  need be computed. The extra work to find  $\gamma$  is one real matrix-vector product and two real vector products.

### A.3 The Singularity Condition

**Theorem 5** *The matrix  $[\lambda_i I - (I - s_i s_i^H)Z]$  is not singular when  $\lambda_i$  is a distinct non-zero eigenvalue and  $Z$  is not defective.*

**Proof:** Since  $Z$  is not defective, Equation A.1 can be used to factor  $Z$  giving

$$[\lambda_i I - S \Lambda S^{-1} + s_i s_i^H S \Lambda S^{-1}].$$

This matrix is non-singular if all the eigenvalues are non-zero. To see the eigenvalues, apply a similarity transformation by post-multiplying by  $S$  and pre-multiplying by  $S^{-1}$ . Since  $Z$  is not defective,  $S$  is invertible, and the eigenvalues are invariant under the invertible similarity transform. After canceling  $S^{-1}S$  terms, where appropriate

$$[\lambda_i I - \Lambda + S^{-1} s_i s_i^H S \Lambda].$$

Consider the term:  $S^{-1} s_i s_i^H S \Lambda$ . This can be resolved as follows

$$S^{-1} s_i = e_i,$$

$$s_i^H S = [\beta_{1,i}, \dots, \beta_{i-1,i}, 1, \beta_{i+1,i}, \dots, \beta_{n,i}]$$

where  $e_i$  is a unit vector in the  $i$ 'th direction and  $\beta_{j,i}$  is the inner product between  $s_j$  and  $s_i^H$  (always  $\leq 1$ ). So

$$S^{-1} s_i s_i^H S \Lambda = \begin{bmatrix} 0 & \dots & 0 & 0 & 0 & \dots & 0 \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & 0 & 0 & \dots & 0 \\ \lambda_1 \beta_{1,i} & \dots & \lambda_{i-1} \beta_{i-1,i} & \lambda_i & \lambda_{i+1} \beta_{i+1,i} & \dots & \lambda_n \beta_{n,i} \\ 0 & \dots & 0 & 0 & 0 & \dots & 0 \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & 0 & 0 & \dots & 0 \end{bmatrix}.$$

Now add this to the two diagonal arrays:  $\lambda_i I - \Lambda$  for the following result:

$$S^{-1} [\lambda_i I - (I - s_i s_i^H)Z] S =$$



$$\begin{bmatrix} \lambda_i - \lambda_1 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & \lambda_i - \lambda_{i-1} & 0 & 0 & \cdots & 0 \\ \lambda_1 \beta_{1,i} & \cdots & \lambda_{i-1} \beta_{i-1,i} & \lambda_i & \lambda_{i+1} \beta_{i+1,i} & \cdots & \lambda_n \beta_{n,i} \\ 0 & \cdots & 0 & 0 & \lambda_i - \lambda_{i+1} & \cdots & 0 \\ \vdots & & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & 0 & \cdots & \lambda_i - \lambda_n \end{bmatrix}. \quad (\text{A.10})$$

By inspection, the matrix is non-singular when all the diagonal elements are non-zero. In fact, the diagonal elements  $(\lambda_i - \lambda_1, \dots, \lambda_i - \lambda_{i-1}, \lambda_i, \lambda_i - \lambda_{i+1}, \dots, \lambda_i - \lambda_n)$  are the eigenvalues. So the necessary and sufficient conditions for non-singularity are:  $\lambda_i \neq 0$  and  $\lambda_j \neq \lambda_i$  for all  $j \neq i$ .

This completes the proof but it is interesting to consider  $\lambda_i = 0$  further. If  $\lambda_i = 0$  the null space of the matrix on the right side of Equation A.10 is  $e_i$ . By reversing the similarity transform, we see that the null space of the original matrix (when  $\lambda_i = 0$ ) is  $s_i$ , the eigenvector of  $Z$  corresponding to  $\lambda_i$ . Since we already know that  $s_i^\perp$  is orthogonal to  $s_i$ , the fact that  $s_i$  is a null space of  $[\lambda_i I - (I - s_i s_i^H)Z]$  makes absolutely no difference in Equation A.7. Therefore, Equation A.8 is still correct if the pseudo inverse (which has columns orthogonal to  $s_i$ ) is used in place of the ordinary inverse.



## Appendix B

### Discrete Conversion

In Chapter 6, the quantities  $\Phi$ ,  $Q$ , and  $R$  were defined for continuous transitions with no discrete events. Methods are developed here for computing  $\Phi$ ,  $Q$ ,  $R$ , and their partial derivatives with respect to a scalar variation in the state differential equation matrix. These results are based on Van Loan's work [VL78].

#### B.1 Problem Statement

Assume a linear, time invariant continuous system described by matrix state equation  $\dot{x}_c = Ax_c + Bx_s$ , where  $x_c$  is the continuous part of the state vector (which is changing) and  $x_s$  is the sample and hold part of the state vector (which is constant). For convenience,  $A$  and  $B$  are grouped into a single matrix:

$$Z = \begin{bmatrix} A & B \\ 0 & 0 \end{bmatrix}.$$

Let  $A$  and  $B$  be a function of a scalar  $\alpha$  such that

$$\frac{\partial Z}{\partial \alpha} = \dot{Z} = \begin{bmatrix} \frac{\partial A}{\partial \alpha} & \frac{\partial B}{\partial \alpha} \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} \dot{A} & \dot{B} \\ 0 & 0 \end{bmatrix}.$$

The desired results are expressions for the matrices  $\Phi$ ,  $Q$ , and  $R$  defined earlier (see Equations 3.3, 6.6, and 6.3). For efficiency, the following abbreviated forms

and their partials will actually be calculated:

$$\Phi(t) = \exp(Z t), \quad (\text{B.1})$$

$$\phi(t) = \exp(A t), \quad (\text{B.2})$$

$$R_c(t) = \int_0^t \phi(s) x_{11}^0 \phi(s)^T ds, \text{ and} \quad (\text{B.3})$$

$$Q(t) = \int_0^t \Phi(s)^T W_{cs} \Phi(s) ds, \quad (\text{B.4})$$

where  $x_{11}^0$  is the continuous process noise covariance matrix and  $W_{cs}$  is the top left corner of the cost weighting matrix (for the continuous and hold states). To obtain the correct form for the Chapter 6 expressions, the rightmost columns and bottom rows must be added to get the correct size matrix.  $R$  should be filled with zeros,  $\Phi$  should be filled with an identity matrix, and  $Q$  should be filled with  $W_0 t$ .

Also, we need the partial of  $\Phi$ ,  $Q$ , and  $R$  with respect to a scalar  $\alpha$ . All these partials should be right filled and bottom filled with zeros to obtain the correct sizes.

## B.2 Van Loan Results

Van Loan [VL78] proved the following: If

$$C = \begin{bmatrix} A_1 & B_1 & C_1 \\ 0 & A_2 & B_2 \\ 0 & 0 & A_3 \end{bmatrix}$$

is a block triangular matrix of constants and

$$e^{(Ct)} = \begin{bmatrix} F_1(t) & G_1(t) & H_1(t) \\ 0 & F_2(t) & G_2(t) \\ 0 & 0 & F_3(t) \end{bmatrix},$$

then:

$$F_j(t) = e^{A_j t}, \quad (\text{B.5})$$

$$G_j(t) = \int_0^t e^{A_j(t-s)} B_j e^{A_{j+1}s} ds, \text{ and} \quad (\text{B.6})$$

$$H_j(t) = \int_0^t e^{A_j(t-s)} C_j e^{A_{j+2}s} ds + \int_0^t \int_0^s e^{A_j(t-s)} B_j e^{A_{j+1}(s-r)} B_{j+1} e^{A_{j+2}r} dr ds. \quad (\text{B.7})$$

### B.3 Main Result

$$\text{Let } C_Q = \begin{bmatrix} -Z^T & W_{cs} & 0 \\ 0 & Z & \dot{Z} \\ 0 & 0 & Z \end{bmatrix} \text{ and } e^{(C_Q t)} = \begin{bmatrix} F_1(t) & G_1(t) & H_1(t) \\ 0 & F_2(t) & G_2(t) \\ 0 & 0 & F_3(t) \end{bmatrix}.$$

Then:

$$\Phi(t) = F_2(t) = F_3(t), \quad (\text{B.8})$$

$$\frac{\partial \Phi(t)}{\partial \alpha} = G_2(t), \quad (\text{B.9})$$

$$Q(t) = F_2^T(t)G_1(t), \text{ and} \quad (\text{B.10})$$

$$\frac{\partial Q(t)}{\partial \alpha} = F_2^T(t)H_1(t) + [F_2^T(t)H_1(t)]^T. \quad (\text{B.11})$$

Parallel results are obtained for  $R$  by replacing  $Z$  with  $A^T$  and  $W_{cs}$  with  $x_{11}^0$ .

$$\text{Let } C_R = \begin{bmatrix} -A & x_{11}^0 & 0 \\ 0 & A^T & \dot{A}^T \\ 0 & 0 & A^T \end{bmatrix} \text{ and } e^{(C_R t)} = \begin{bmatrix} F_1(t) & G_1(t) & H_1(t) \\ 0 & F_2(t) & G_2(t) \\ 0 & 0 & F_3(t) \end{bmatrix}.$$

Then:

$$R_c(t) = F_2^T(t)G_1(t), \text{ and} \quad (\text{B.12})$$

$$\frac{\partial R_c(t)}{\partial \alpha} = F_2^T(t)H_1(t) + [F_2^T(t)H_1(t)]^T. \quad (\text{B.13})$$

If only  $\frac{\partial \Phi(t)}{\partial \alpha}$  is required, it can be found from

$$C_\Phi = \begin{bmatrix} Z & \dot{Z} \\ 0 & Z \end{bmatrix}$$

so

$$e^{(C_\Phi t)} = \begin{bmatrix} e^{Zt} & \frac{\partial e^{Zt}}{\partial \alpha} \\ 0 & e^{Zt} \end{bmatrix}. \quad (\text{B.14})$$

When partials are not required,  $Q$  and  $R$  are found using the top left two-by-two blocks of  $C_Q$  and  $C_R$  in Equations B.10 and B.12. This simplifies the calculations for cases without variable parameters in the  $A$  or  $B$  matrices.

### B.4 Proofs

Equations B.8, B.10, and B.12 follow directly from the definitions and from Van Loan's results in Equations B.5 and B.6.

### B.4.1 $\Phi$ Partial

$\Phi$  has a series expansion:

$$\Phi = I + Zt + \frac{Z^2 t^2}{2!} + \frac{Z^3 t^3}{3!} + \dots \quad (\text{B.15})$$

where Equation B.15 is the ordinary expansion of a matrix exponential.

The series expansion of the partial of  $\Phi$  with respect to  $\alpha$  is found by differentiating Equation B.15 term by term:

$$\frac{\partial \Phi}{\partial \alpha} = \dot{Z}t + \frac{(\dot{Z}Z + Z\dot{Z})t^2}{2!} + \frac{(\dot{Z}Z^2 + Z\dot{Z}Z + Z^2\dot{Z})t^3}{3!} + \dots \quad (\text{B.16})$$

Compare this with:

$$\exp\left(\begin{bmatrix} Z & \dot{Z} \\ 0 & Z \end{bmatrix} t\right).$$

Expand the first few terms of the exponential series for:

$$\begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix} + \begin{bmatrix} Z & \dot{Z} \\ 0 & Z \end{bmatrix} t + \begin{bmatrix} Z^2 & (Z\dot{Z} + \dot{Z}Z) \\ 0 & Z^2 \end{bmatrix} \frac{t^2}{2!} + \dots$$

The top right block in the first few terms certainly match the desired result. We will prove that the top right block is the desired result by induction. Assume the the matrix part of the  $k$ 'th term of the series expansion is given by:

$$\begin{bmatrix} Z^k & (Z^{k-1}\dot{Z} + Z^{k-2}\dot{Z}Z + \dots + \dot{Z}Z^{k-1}) \\ 0 & Z^k \end{bmatrix}.$$

Then the next  $(k+1)$  term will be:

$$\begin{aligned} & \begin{bmatrix} Z^k & (Z^{k-1}\dot{Z} + Z^{k-2}\dot{Z}Z + \dots + \dot{Z}Z^{k-1}) \\ 0 & Z^k \end{bmatrix} \begin{bmatrix} Z & \dot{Z} \\ 0 & Z \end{bmatrix} \\ &= \begin{bmatrix} Z^{k+1} & (Z^k\dot{Z} + Z^{k-1}\dot{Z}Z + \dots + \dot{Z}Z^k) \\ 0 & Z^{k+1} \end{bmatrix}. \end{aligned}$$

So, we have proved by induction that:

$$\exp\left(\begin{bmatrix} Z & \frac{\partial Z}{\partial \alpha} \\ 0 & Z \end{bmatrix} t\right) = \begin{bmatrix} \exp(Zt) & \frac{\partial \exp(Zt)}{\partial \alpha} \\ 0 & \exp(Zt) \end{bmatrix}. \quad (\text{B.17})$$

Compare this with Equation B.6 for  $A_1 = A_2 = Z$ ,  $B_1 = \dot{Z}$ , and  $C_1 = B_2 = A_3 = 0$  and we can also conclude that

$$\frac{\partial e^{Zt}}{\partial \alpha} = \int_0^t e^{Z(t-s)} \frac{\partial Z}{\partial \alpha} e^{Zs} ds = \int_0^t e^{Z(t-s)} \dot{Z} e^{Zs} ds. \quad (\text{B.18})$$

At this point, direct comparison with Equation B.6 confirms Equation B.9. So, Equations B.9 and B.14 are proved.

#### B.4.2 $Q$ and $R$ Partialals

Equations B.11 and B.13 are duals so only one of them need be proved. Equation B.11 will be proved since its notation closely parallels the previous section.

Begin by taking the partial differential of the  $Q(t)$  definition with respect to  $\alpha$ :

$$\frac{\partial Q(t)}{\partial \alpha} = \int_0^t \Phi(s)^T W_{cs} \frac{\partial \Phi(s)}{\partial \alpha} ds + \int_0^t \frac{\partial \Phi(s)^T}{\partial \alpha} W_{cs} \Phi(s) ds.$$

Now, substitute Equation B.18 for the partials to obtain:

$$\begin{aligned} \frac{\partial Q(t)}{\partial \alpha} &= \int_0^t \Phi(s)^T W_{cs} \int_0^s e^{Z(s-r)} \dot{Z} e^{Zr} dr ds \\ &+ \int_0^t \int_0^s e^{Z^T r} \dot{Z}^T e^{Z^T(s-r)} dr W_{cs} \Phi(s) ds. \end{aligned} \quad (\text{B.19})$$

Recognize that the second integral is the transpose of the first and that the first is equal to  $F_3^T(t)H_1(t)$  from Equation B.7 with  $C = C_Q$ . This completes the proof.





# Appendix C

## STM Continuity

### C.1 The Phase Condition

Consider an asynchronous hybrid linear system. Such a system includes a linear, time invariant continuous part with two or more asynchronous sample processes. A basic time period is established (perhaps arbitrarily) and this establishes a time window of interest. Let all the sample schedules be phase locked to the BTP except one that is allowed to slide with respect to the BTP. Figure C.1 illustrates this setup.

Suppose the state transition matrix (STM) for the BTP is expanded as:

$$\Psi = \Psi_k S_{k-1} \Psi_{k-1} \cdots \Psi_3 S_2 \Psi_2 S_1 \Psi_1 S_0$$

where  $\Psi_1$  is the STM from the start of the BTP to the first event in the sliding sequence,  $S_1$  is the STM for the first discrete event in the sliding sequence, etc.

Now, allow all the events in the sliding sequence to slip  $\Delta t$  into the future where  $\Delta t$  is small enough that:

1. None of the events in the sliding sequence leave or enter the BTP, and
2. None of the events in the sliding sequence meet or reverse order with events in any of the other sequences.

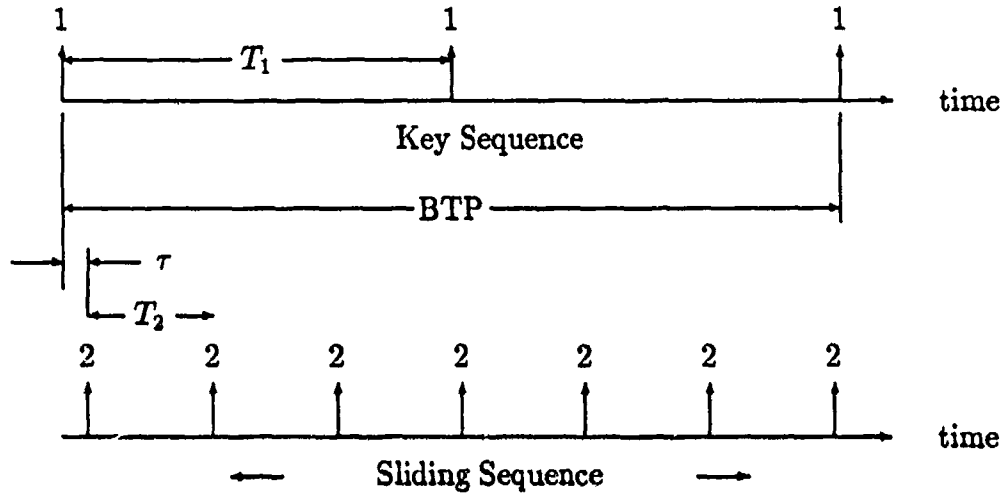


Figure C.1: Phase Relationships

Then, the BTP STM for the slipped sequence can be written as

$$\Psi(\text{slip}) = \Psi_k \Phi(-\Delta t) S_{k-1} \Phi(\Delta t) \Psi_{k-1} \cdots \Psi_2 \Phi(-\Delta t) S_1 \Phi(\Delta t) \Psi_1$$

where  $\Phi(\Delta t) = \exp(Z \Delta t)$  is the continuous state transition matrix for an elapsed time  $\Delta t$ .

Since  $\Phi(\Delta t)$  is a continuous, analytic function of  $t$  (a matrix polynomial in fact), and all the other matrices are constant,  $\Psi$  is clearly a continuous analytic function of the slip.

Since this slip is synonymous with the phase ( $\tau_i$ ) of one of the asynchronous sequences, the BTP STM is seen to be continuous and analytic with respect to the phase vector as long as the two conditions are met.

The total phase space can be partitioned into regions where  $\Psi$  is continuous simply by finding the boundaries. So the problem is to identify all the delays  $\tau$  where  $\tau_i \in [0, T_i)$  such that

$$\lim_{\epsilon \rightarrow 0} \Psi(\tau_i - \epsilon) \neq \Psi(\tau_i + \epsilon)$$

where  $T_i$  is the period of the asynchronous schedule and  $\Psi(\tau)$  is the state transition matrix for one BTP with phasing  $\tau$ . The easy way to do this is to identify each phase value where either of the two conditions occurs. This approach may find

discontinuity boundaries where the STM is actually continuous, but that is not a problem.

## C.2 Boundary Conditions

There are two conditions that can produce an STM discontinuity: first, when a discrete event enters or leaves the current BTP, and second, when discrete events reverse order. These will be considered one at a time.

For the rest of the discussion, it is assumed that there is only one asynchronous sequence with period  $T$ .

### C.2.1 BTP Truncation Cases

Let  $\rho$  be the remainder when the BTP is divided by  $T$ . That is to say,  $\text{BTP} = kT + \rho$ , where  $k$  is an integer and  $\rho < T$ . Then, the BTP contains  $k$  complete periods of the asynchronous schedule when  $\tau \in (0, \rho)$  and only  $k - 1$  periods when  $\tau \in (\rho, T)$ . It is unreasonable to expect continuity when the number of complete cycles in the BTP changes, so  $\tau_i$  is a candidate discontinuity point if:

$$\tau_i = \rho - \xi_k \text{ or } \tau_i = T - \xi_k \rho \quad (\text{C.1})$$

where  $\xi_k$  is the time from the start of the asynchronous schedule to the  $k$ 'th discrete event in that schedule.

### C.2.2 Commutivity Cases

Suppose there is a value  $\tau_i$  such that a discrete event in synchronous schedule coincides (in time) with a discrete event in the asynchronous schedule. Furthermore, suppose the coincidence occurs during the BTP at  $\tau = \tau_i$ . Then the order of the two discrete state transition matrices will be reversed in  $\Psi(\tau_i - \epsilon)$  and  $\Psi(\tau_i + \epsilon)$ . If these two discrete state transition matrices do not commute, then  $\tau_i$  is a discontinuity point.

This condition can be described mathematically as:

$$k_1 T_1 + \xi_{1j} \leq BTP, \text{ and } k_1 T_1 + \xi_{1j} = \tau_i + k_2 T_2 + \xi_{2k} \text{ and } D_{1j} D_{2k} \neq D_{2k} D_{1j} \quad (C.2)$$

where

$k_1, k_2$  are any integers,

$T_1$  is the period of the synchronous schedule,

$T_2$  is the period of the asynchronous schedule,

$\xi_{1j}$  is the time from the start of the synchronous schedule to the  $j$ 'th discrete event in that schedule,

$\xi_{2k}$  is the time from the start of the asynchronous schedule to the  $k$ 'th discrete event in that schedule,

$D_{1j}$  is the state transition matrix for the  $j$ 'th discrete event in the synchronous schedule, and

$D_{2k}$  is the state transition matrix for the  $k$ 'th discrete event in the asynchronous schedule.

### C.2.3 Recap

Equation C.1 and Equation C.2 define the phase of all potential discontinuities in the current BTP state transition matrix. The actual values can be found by a simple but tedious exhaustive search.

Discontinuities in the previous BTP state transition matrix can be found by substituting  $T_2 - \tau$  for  $\tau$  and working backwards from the end of the period and BTP. The union of discontinuity times for the current and previous BTP's is a candidate discontinuity set for the eigenvector function  $S_i S_{i-1}^{-1}$ .

# Appendix D

## Theorem Proofs

This chapter provides proofs of the four theorems used in Chapter 6.

### D.1 Notation and Definitions

The following notation is common to the proofs.

- $x_c(t)$  = vector of the continuous-time states
- $x_s(t)$  = vector of the sample-and-hold states
- $x_d(t)$  = vector of the discrete-time states
- $X(t)$  = the full state vector  $[x_c^T \ x_s^T \ x_d^T]^T$
- $XX(t)$  = full state vector covariance:  $E\{X(t)X^T(t)\}$
- $w_c(t)$  = vector of the continuous-time process noise
- $w_s(t)$  = vector of sampling errors
- $w_d(t)$  = vector of discrete state update noise
- $n_c(t)$  = augmented vector of continuous process noise =  $[w_c^T \ 0]^T$
- $n_d(t)$  = vector of discrete process noise =  $[0 \ w_s^T \ w_d^T]^T$

- $N(t)$  = total process noise vector
- $W_0(t)$  = state error weighting matrix for cost function

## D.2 Theorems and Proofs

**Theorem 1** Let  $x_{cs}$  be the partial state vector:  $[x_c^T x_s^T]^T$  such that

$$\dot{x}_{cs} = [AB] x_{cs} + n_{cs}$$

where  $n_{cs}$  is white continuous process noise with covariance:

$$E\{n_{cs}(s)n_{cs}^T(t)\} = \begin{bmatrix} x_{11}^0 & 0 \\ 0 & 0 \end{bmatrix} \delta(s-t) = X_{cs}^0 \delta(s-t)$$

where  $\delta(\cdot)$  is a unit impulse at zero.

If  $xx(t) \equiv E\{x_{cs}(t)x_{cs}^T(t)\}$  and there are no discrete transitions between  $t = 0$  and  $t = t_1$ , then

$$xx(t_1) = \phi(t_1)xx(0)\phi(t_1)^T + R_c(t_1)$$

where  $\phi(s) \equiv \exp([AB]s)$  and

$$R_c(t_1) = \int_0^{t_1} \phi(s)X_{cs}^0\phi(s)^T ds.$$

**Proof:** From linear system theory, for  $t \leq t_1$ ,

$$x_{cs}(t) = \phi(t)x_{cs}(0) + \int_0^t \phi(s)n_{cs}(s) ds.$$

So,

$$\begin{aligned} xx(t) = E\{ & \left( \phi(t)x_{cs}(0) + \int_0^t \phi(r)n_{cs}(r) dr \right) \\ & \left( x_{cs}^T(0)\phi^T(t) + \int_0^t n_{cs}^T(s)\phi^T(s) ds \right) \}. \end{aligned}$$

The process noise is uncorrelated with the initial state so  $E\{x_{cs}(s)n_{cs}^T(t)\} = 0$  for all  $s < t$ . Therefore, the cross products are zero. Evaluating the remaining terms:

$$\begin{aligned} E\{\phi(t)x_{cs}(0)x_{cs}^T(0)\phi^T(t)\} &= \phi(t)E\{x_{cs}(0)x_{cs}^T(0)\}\phi^T(t) \\ &= \phi(t)xx(0)\phi^T(t) \end{aligned}$$

and

$$\begin{aligned}
 & E\left\{\int_0^t \phi(r) n_{cs}(r) dr \int_0^t n_{cs}^T(s) \phi^T(s) ds\right\} \\
 &= E\left\{\int_0^t \int_0^t \phi(r) n_{cs}(r) n_{cs}^T(s) \phi^T(s) dr ds\right\} \\
 &= \int_0^t \phi(r) \int_0^t E\{n_{cs}(r) n_{cs}^T(s)\} \phi^T(s) ds dr \\
 &= \int_0^t \phi(r) X_{cs}^0 \phi^T(r) dr.
 \end{aligned}$$

Therefore,

$$xx(t) = \phi(t)xx(0)\phi^T(t) + \int_0^t \phi(r) X_{cs}^0 \phi^T(r) dr$$

**Theorem 2** Let  $X = [x_c^T x_s^T x_d^T]^T$  be the full state vector. Let  $\Psi_d$  be a discrete state transition matrix where

$$X(t^+) = \Psi_d X(t^-) + n_d(t)$$

and where  $n_d(t)$  is white discrete process noise with covariance  $R_d$ .

If  $XX(t) \equiv E\{X(t)X^T(t)\}$  and the discrete transition  $\Psi_d$  occurs at time  $t_0$ , then

$$XX(t_0^+) = \Psi_d XX(t_0^-) \Psi_d^T + R_d.$$

Note: let  $R_d = [R_{ij}]$ . Then  $R_{ij}$  is zero unless  $\Psi_d$  updates state  $i$  or state  $j$ .

**Proof:** From the state equation,

$$XX(t_0^+) = E\{(\Psi_d X(t^-) + n_d)(X^T(t^-) \Psi_d^T + n_d^T)\}.$$

But  $E\{X(t^-)n_d^T(t_0)\} = 0$  and  $E\{n_d(t_i)n_d^T(t_j)\} = \delta_{ij} R_d$ .

$$\text{So, } XX(t_0^+) = \Psi_d E\{X(t^-)X^T(t^-)\} \Psi_d^T + E\{n_d n_d^T\} = \Psi_d XX(t^-) \Psi_d^T + R_d.$$

**Theorem 3** Let  $X$  be the full state vector:  $[x_c^T x_s^T x_d^T]^T$  and

$$\dot{X} = G(t)X + N(t).$$

$G(t)$  is stable with a corresponding state transition matrix  $\Psi(t_b, t_a)$ .  $N(t)$  is white process noise and  $E\{N(r)N^T(s)\} = \delta(r-s)R_n(s)$ .

Let  $XX(t) \equiv E\{X(t)X^T(t)\}$ ,  $t_0 > t_{-1} > t_{-2} \dots$ , and  $\lim_{n \rightarrow -\infty} (t_n) = -\infty$ .

Then if  $R_i \equiv E\{X(t_i)X^T(t_i)\}$  when  $E\{X(t_{i-1})X^T(t_{i-1})\} = 0$  (i.e. the covariance growth from  $t_{i-1}$  to  $t_i$ ) then

$$XX(t_0) = \sum_{i=0}^{-\infty} \Psi(t_0, t_i) R_i \Psi(t_0, t_i)^T.$$

**Proof:** Let  $t_{-k}$  be some time in the past. Then:

$$X(t_0) = \Psi(t_0, t_{-k})X(t_{-k}) + \int_{t_{-k}}^{t_0} \Psi(t_0, r)N(r)dr$$

or

$$X(t_0) = \Psi(t_0, t_{-k})X(t_{-k}) + \sum_{i=0}^{-(k-1)} \int_{t_{i-1}}^{t_i} \Psi(t_0, r)N(r)dr.$$

So,

$$\begin{aligned} XX(t_0) &= E\{\Psi(t_0, t_{-k})X(t_{-k})X^T(t_{-k})\Psi^T(t_0, t_{-k})\} \\ &+ E\left\{\left(\sum_{i=0}^{-(k-1)} \int_{t_{i-1}}^{t_i} \Psi(t_0, r)N(r)dr\right)X^T(t_{-k})\Psi^T(t_0, t_{-k})\right\} \\ &+ E\{\Psi(t_0, t_{-k})X(t_{-k})\left(\sum_{j=0}^{-(k-1)} \int_{t_{j-1}}^{t_j} N^T(s)\Psi^T(t_0, s)ds\right)\} \\ &+ E\left\{\sum_{i=0}^{-(k-1)} \int_{t_{i-1}}^{t_i} \Psi(t_0, r)N(r)dr \sum_{j=0}^{-(k-1)} \int_{t_{j-1}}^{t_j} N^T(s)\Psi^T(t_0, s)ds\right\} \\ XX(t_0) &= \Psi(t_0, t_{-k})E\{X(t_{-k})X^T(t_{-k})\}\Psi^T(t_0, t_{-k}) \\ &+ \sum_{i=0}^{-(k-1)} \int_{t_{i-1}}^{t_i} \Psi(t_0, r)E\{N(r)X^T(t_{-k})\}dr\Psi^T(t_0, t_{-k}) \\ &+ \Psi(t_0, t_{-k})\sum_{j=0}^{-(k-1)} \int_{t_{j-1}}^{t_j} E\{X(t_{-k})N^T(s)\}\Psi^T(t_0, s)ds \\ &+ \sum_{i=0}^{-(k-1)} \sum_{j=0}^{-(k-1)} \int_{t_{i-1}}^{t_i} \Psi(t_0, r) \int_{t_{j-1}}^{t_j} E\{N(r)N^T(s)\}\Psi^T(t_0, s)dsdr. \end{aligned}$$

Observe two things. First, the system is stable so  $E\{X(t)X^T(t)\}$  is finite and  $\lim_{k \rightarrow \infty} \Psi(t_0, t_{-k})E\{X(t_{-k})X^T(t_{-k})\}\Psi^T(t_0, t_{-k}) = 0$ . Second, the noise is uncorrelated with prior noise and with prior states. Therefore, the cross contribution from disjoint intervals is zero, i.e.  $E\{W(s)W^T(t)\} = 0$  when  $s$  and  $t$  belong to different intervals. Therefore, all the cross product ( $i \neq j$ ) terms are zero.



So

$$\begin{aligned}\lim_{k \rightarrow \infty} XX(t_0) &= \sum_{i=0}^{\infty} \int_{t_i}^{t_{i-1}} \Psi^T(t_0, r) \int_{t_i}^{t_{i-1}} E\{N(r)N^T(s)\} \Psi^T(t_0, s) ds dr \\ &= \sum_{i=0}^{\infty} \int_{t_{i-1}}^{t_i} \Psi(t_0, r) R_n(r) \Psi^T(t_0, r) dr.\end{aligned}$$

Each  $\Psi(t_0, t)$  can be factored as  $\Psi(t_0, t_i)\Psi(t_i, t)$  so

$$\lim_{k \rightarrow \infty} XX(t_0) = \sum_{i=0}^{\infty} \Psi(t_0, t_i) \left[ \int_{t_{i-1}}^{t_i} \Psi(t_i, r) R_n(r) \Psi^T(t_i, r) dt \right] \Psi^T(t_0, t_i).$$

But the term in brackets is just  $R_.$ , as defined above.

**Theorem 4** Let  $X$  be the state vector:  $[x_c^T x_s^T x_d^T]^T$  such that

$$\dot{X} = G X + N$$

where  $G$  is constant and  $N$  is white stationary process noise with covariance

$$X_c^0 = \begin{bmatrix} x_{11}^0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Let  $XX(t_i) \equiv E\{X(t_i)X^T(t_i)\}$  and  $W_0$  be a symmetric matrix.

If there are no discrete transitions between time=0 and time= $t$ , then

$$\begin{aligned}J(t, 0) &\equiv E\left\{\int_0^t X^T(r) W_0 X(r) dr\right\} \\ &= \tilde{\Sigma}\left(XX(0) * \int_0^t \Psi(r)^T W_0 \Psi(r) dr\right) \\ &\quad + \tilde{\Sigma}\left(X_c^0 * \int_0^t \int_0^r \Psi^T(s) W_0 \Psi(s) ds dr\right)\end{aligned}$$

where  $J(t, 0)$  is part of the cost integral for the segment  $(0, t)$ , "\*" denotes element-by-element matrix multiplication,  $\tilde{\Sigma}$  denotes the algebraic sum of the matrix elements, and  $\Psi(t) \equiv \exp\left(\begin{bmatrix} A & B & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} t\right).$

**Proof:** From linear system theory,

$$X(t) = \Psi(t)X(0) + \int_0^t \Psi(s)N(s)ds.$$

So,

$$\begin{aligned} E\left\{\int_0^t X(s)^T W_0 X(s) ds\right\} \\ = E\left\{\int_0^t \left(X^T(0)\Psi^T(r) + \int_0^r N^T(s)\Psi^T(s)ds\right) W_0 \right. \\ \left. \left(\Psi(r)X(0) + \int_0^r \Psi(u)N(u)du\right) dr\right\}. \end{aligned}$$

But  $X(0)$  and  $N(t)$  are uncorrelated if  $t > 0$  so the expected value of the cross products is zero. Therefore,

$$\begin{aligned} E\left\{\int_0^t X(s)^T W_0 X(s) ds\right\} \\ = E\left\{\int_0^t X^T(0)\Psi^T(s)W_0\Psi(s)X(0)ds\right\} \\ + E\left\{\int_0^t \left(\int_0^r N^T(s)\Psi^T(s)ds\right) W_0 \left(\int_0^r \Psi(u)N(u)du\right) dr\right\}. \end{aligned}$$

In the first term, the constant  $X(t_0)$  can be brought outside the integral.

$$\begin{aligned} E\left\{\int_0^t X^T(0)\Psi^T(s)W_0\Psi(s)X(0)ds\right\} \\ = E\left\{X^T(0)\left(\int_0^t \Psi^T(s)W_0\Psi(s)ds\right)X(0)\right\} \\ = \hat{\Sigma}\left(XX(t_0) * \int_0^t \Psi^T(s)W_0\Psi(s)ds\right). \end{aligned}$$

Likewise, the process noise is stationary and the expected value operation will yield a constant. Therefore, it too can be moved outside the inner integrals as follows. For the other term,

$$\begin{aligned} E\left\{\int_0^t \left(\int_0^r N^T(s)\Psi^T(s)ds\right) W_0 \left(\int_0^r \Psi(u)N(u)du\right) dr\right\} \\ = E\left\{\int_{r=0}^t \int_{s=0}^r \int_{u=0}^r N^T(s)\Psi^T(s)W_0\Psi(u)N(u)du ds dr\right\} \\ = E\left\{\int_{r=0}^t \int_{s=0}^r \left(\int_{u=0}^r \hat{\Sigma}(N(u)N^T(s)) * (\Psi^T(s)W_0\Psi(u))du\right) ds dr\right\} \end{aligned}$$

$$\begin{aligned}
&= \int_{r=0}^t \int_{s=0}^r \left( \int_{u=0}^r \tilde{\Sigma} E\{N(u)N^T(s)\} \star (\Psi^T(s)W_0\Psi(u)) du \right) ds dr \} \\
&= \int_{r=0}^t \int_{s=0}^r \tilde{\Sigma} X_c^0 \star (\Psi^T(s)W_0\Psi(u)) ds dr \} \\
&= \tilde{\Sigma} X_c^0 \star \int_{r=0}^t \int_{s=0}^r \Psi^T(s)W_0\Psi(s) ds dr \}
\end{aligned}$$

since  $x_c^0$  is a constant which can be brought outside the integrals.



## Appendix E

### Computer Codes

The algorithms for stability analysis and for asynchronous design were implemented in PC-MATLAB and executed on an IBM-PC clone. The MATLAB code is close to ordinary mathematical expressions and it is commented so these routines should be useful without additional documentation. These same codes will run on some Unix machines with PRO-MATLAB (a Unix version of MATLAB). The Unix hosts (e.g. Sun 3) are much faster than the PC.

To use this software, load all the ".m" files in some directory where PC-MATLAB will run. Then, run "matlab" at the operating system prompt. Then, command either "design" or "analyze" at the MATLAB prompt ">>". Finally, answer the questions (following each answer with a RETURN).

To run your own examples, use buildtla.m or bulddin.m as a guide to create a new buildxxx.m file for the system you want to investigate. Name your file buildtla.m or bulddin.m and run as before. Alternatively, use a new name for your buildxxx.m file and patch the routines: analyze.m, design.m, and var\_opt2.m (xxxtheta) to use your file instead of bulddin.m or buildtla.m.

The subroutines are arranged in three groups:

- Utility Subroutines (may be called by analysis or design routines)
- Analysis Subroutines
- Design Subroutines

## E.1 Utility Routines

### E.1.1 Build Two-Link Arm System: buildtla.m

```

function [AB,DS,seqptr,Theta,varptr,w0,x0,BTP,keyseq]=buildtla
%
% Buildtla loads the initial system description for Berg's two link-arm
% plant/controller. Final data from Berg's Case 1 is used.
%
% Figure and table numbers refer to Berg's Thesis.
%
%-----
%
% The states are:      x(1): hub angle, theta           (see fig 6.1)
%                    x(2): hub angle rate, theta dot
%                    x(3): tip position, delta
%                    x(4): tip speed, delta dot
%                    x(5): sample & hold 1 (h1)           (see fig 6.2)
%                    x(6): sample & hold 2 (h2)
%                    x(7): sample & hold 3 (h3)
%                    x(8): slow discrete control state, c1
%                    x(9): fast discrete control state, c2
%
%-----
%
% Set sampling times (ref table 6.3)

fprintf('\nloading two link arm plant/controller.\n');

keyseq=1;           % the sequence that is the basis of the BPT
STP=1/35.556;       % short time period (synchronous)
LTP=8*STP;          % long time period
%STP=LTP/(2*pi);    % short time period (asynchronous)
BTP=LTP;            % basic time period

%-----
%
% 'seqptr' is an array that defines the sampling sequences. Seqptr has
% a row for each synchronous sample sequence. Each entry in a row
% represents a continuous-time or discrete-time state transition.
% Positive entries represent continuous transitions for that duration.
% Negative integers indicate discrete-time transitions described by the
% corresponding block of DS (i.e. -1 and -2 for this problem). A zero
% (which may be required to fill the rectangular array) represents no
% transition. The first element in a row is the first element in the
% sequence (chronological) which must indicate a discrete transition.

```

% The period of any sequence is the sum of the non-negative elements in  
 % the corresponding row of "seqptr".

```
seqptr=[-1,LTP;           % slow sample cycle, starts with D1, period=LTP
        -2,STP];         % fast sample cycle, starts with D2, period=STP
```

%-----  
 %  
 % Theta is a vector of the variable parameters. These elements  
 % must be the same as their definitions in varptr. (ref table 6.5)

```
Theta=[-0.48499;          % -(alpha-1)
        11.297;           % beta-11
        0.39316;          % beta-12
        -13.483;          % gamma-11
        1.0708;           % gamma-12
        -0.55327;         % -(alpha-2)
        0.097632;         % beta-21
        13.439;           % beta-22
        -0.12129;         % gamma-21
        -16.865 ];        % gamma-22
```

%-----  
 %  
 % "varptr" is an array that defines each of the variables in "Theta".  
 % Each "varptr" row contains (type, row, column) where:  
 %     type = 0 denotes the AB matrix  
 %     type = j (j = positive integer) denotes the j'th block of DS.  
 % The i'th row of "varptr" corresponds to the i'th variable in "Theta"  
 % which belongs in (row,column) of the indicated matrix.

```
varptr=[ 1,8,8;           % alpha-1
        1,5,1;           % beta-11
        1,5,3;           % beta-12
        1,5,8;           % gamma-11
        1,5,9;           % gamma-12
        2,9,9;           % alpha-2
        1,6,1;           % beta-21
        2,7,3;           % beta-22
        1,6,8;           % gamma-21
        2,7,9];          % gamma-22
```

%-----  
 %  
 % AB is a concatenation of the usual, continuous-time  $\dot{A}$  and  $-B$   
 % matrices ( $AB=[A,-B]$ ), where  $\dot{x} = A x + B u$  which is zero-filled  
 % for squareness. Negative feedback is achieved by negating the B

% matrix. All sample-and-hold states are considered to be inputs so B  
 % has a column for each hold state and these columns are sequenced like  
 % the sample-and-hold states. An all-zero column indicates the  
 % corresponding hold state doesn't drive the continuous states. If  
 % the output of two holds is summed to drive a physical input, B will  
 % have duplicate columns. This approach allows the continuous state  
 % transition matrix for time "t" (with no discrete events during t)  
 % to be computed as:

```
%
%
%           Psi(t)= [expm(AB*t), 0; 0, eye(nxd)]
%
```

% where nxd = number of purely discrete-time states.

% In seqptr, AB (a continuous-time transition) is indicated by a  
 % positive entry.

% In varptr, AB elements are indicated by a zero in column 1.

```
%
%
AB= [0, 1, 0, 0, 0, 0, 0, 0; % ref eq 6.4-6.10
      0, 0, 0, 0, -(2.3684), -(-22.934), -(-22.934); % and table 6.1
      0, 0, 0, 1, 0, 0, 0;
      0, 0, 0, 0, -(-1.1428), -(121.59), -(121.59);
      0, 0, 0, 0, 0, 0, 0; % zero fill row
      0, 0, 0, 0, 0, 0, 0; % zero fill row
      0, 0, 0, 0, 0, 0, 0]; % zero fill row
```

```
%-----
%
% "DS" is a concatenation of all the discrete and sample state
% transition matrices (i.e. DS=[d1,d2,...s1,s1,...]) in any order. For
% the TLA problem, d1 is the slow discrete transition and d2 is the
% fast state transition. The fixed elements are loaded now. Update is
% called later (by "check") to load the variable elements (Theta) into
% DS and AB.
```

```
DS=[eye(9) eye(9)];
DS(5,5)=0; DS(6,6)=0; DS(8,8)=0; DS(8,1)=1;
DS(7,16)=0; DS(9,18)=0; DS(9,12)=1;
```

```
%-----
%
% Define cost weighting parameters for optimization
```

% state error weighting coefficients for cost function

```
w0=[21 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0
     0 0 1850 0 0 0 0 0 0]
```



```

0      0      0      0      0      0      0      0      0
0      0      0      0      1      0      0      0      0
0      0      0      0      0      69.44  69.44  0      0
0      0      0      0      0      69.44  69.44  0      0
0      0      0      0      0      0      0      0      0
0      0      0      0      0      0      0      0      0 ];

% process noise covaraence (future use)

x0=[0      0      0      0      0      0      0      0      0
0      915.48 0      -2976.4 0      0      0      0      0
0      0      0      0      0      0      0      0      0
0      -2976.4 0      14874.1 0      0      0      0      0
0      0      0      0      0      0      0      0      0
0      0      0      0      0      0      0      0      0
0      0      0      0      0      0      0      0      0
0      0      0      0      0      0      0      0      0
0      0      0      0      0      0      0      0      0 ];

% special code to check sensitivity to semi-definite X0 & W0
%w0=w0+1*eye(w0);
%x0=x0+1*eye(w0);
% end of special test code

%-----

fprintf('          Load complete. Berg case 1 data used.\n\n');
return

% end of the buildtla function $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

```

### E.1.2 Build Double Integrator System: builddin.m

```

function [AB,DS,seqptr,Theta,varptr,w0,x0,BTP,keyseq]=builddin
%
% Builddin loads initial system description for the double integrator
% plant/controller. ***** fourth-order version *****
%
%-----
%
% The states are:      x(1): first integrator output
%                      x(2): second integrator output
%                      x(3): sample and hold output (proportional)
%                      x(4): sample and hold output (rate)
%

```

```
fprintf('\nLoading for 4th order double integrator/controller.\n');
```

```
%-----
```

```
keyseq=1;      % the sequence that is the basis of the BPT
%keyseq=2;     % the sequence that is the basis of the BPT
```

```
STP=1.0;      % first time period
%STP=1.1;     % first time period
%LTP=1.0;     % other time period
%LTP=1.1;     % other time period
LTP=.9;       % other time period
```

```
%BTP=STP;     % basic time period
%BTP=LTP;     % basic time period
BTP=9;        % basic time period
```

```
%-----
```

```
% Theta is a vector of the variable parameters. The order of these
% elements must be the same as their definitions in varptr.
```

```
Theta=[0.5;    % +(c1)    % so-called nominal gains
       0.5;    % +(c2)
       2.0];   % +(c3)
```

```
% Theta=[1.121;    % +(c1)    % optimal asynchronous gains
%      0.499;      % +(c2)
%      1.628];     % +(c3)
```

```
% Theta=[0.666;    % +(c1)    % optimal synchronous gains
%      0.115;      % +(c2)
%      1.025];     % +(c3)
```

```
%-----
```

```
%
% "seqptr" is an array that defines the sampling sequences. Seqptr has
% a row for each synchronous sample sequence. Each entry in a row
% represents a continuous-time or discrete-time state transition.
% Positive entries represent continuous transitions for that duration.
% Negative integers indicate discrete-time transitions described by the
% corresponding block of DS (i.e. -1 and -2 for this problem). A zero
% (which may be required to fill the rectangular array) represents no
% transition. The first element in a row is the first element in the
% sequence (chronological) which must indicate a discrete transition.
% The period of any sequence is the sum of the non-negative elements in
% the corresponding row of "seqptr".
```

```
seqptr=[-1,STP,0;    % proportional feedback event
```

```

-2,-3,LTP];      % rate feedback event

%-----
%
% "varptr" is an array that defines each of the variables in "Theta".
% Each "varptr" row contains (type, row, column) where:
%     type = 0 denotes the AB matrix
%     type = j (j = positive integer) denotes the j'th block of DS.
% The i'th row of "varptr" corresponds to the i'th variable in "Theta"
% which belongs in (row,column) of the indicated matrix.

varptr=[ 1,3,2;      % c1
        2,4,2;      % c2
        3,4,4 ];    % c3

%-----
%
% AB is a concatenation of the usual, continuous-time +A and -B
% matrices (AB=[A,-B], where  $\dot{x} = A x + B u$ ) which is zero-filled for
% squareness. Negative feedback is achieved by negating the B matrix.
% All sample-and-hold states are considered to be inputs so B has a
% column for each hold state and these columns are sequenced like the
% sample-and-hold states. An all-zero column indicates the
% corresponding hold state doesn't drive the continuous states. If
% the output of two holds is summed to drive a physical input, B will
% have duplicate columns. This approach allows the continuous state
% transition matrix for time "t" (with no discrete events during t) to
% be computed as:
%
%     Psi(t)= [expm(AB*t), 0; 0, eye(nxd)]
%
% where nxd = number of purely discrete-time states.
%
% In seqptr, AB (a continuous-time transition) is indicated by a
% positive entry.
% In varptr, AB elements are indicated by a zero in column 1.
%
AB= [0, 0, -1, -1,
     1, 0,  0,  0,
     0, 0,  0,  0,
     0, 0,  0,  0];

%-----
%
% "DS" is a concatenation of all the discrete and sample state
% transition matrices (i.e. DS=[d1,d2,...,s1,s1,...]) in any order.
% For the DINT problem, position feedback discrete state transition and

```

```
% d2 is the rate feedback state transition. The fixed elements are
% loaded now. Update is called later to load the variable elements
% (theta) into DS and AB.
```

```
DS=[eye(4) eye(4) eye(4)];
DS(3,1:4)=[0 1 0 0]; % DS(3,1:4)= [0 c1 0 0] after update
DS(4,5:8)=[0 1 -1 0]; % DS(4,5:8)= [0 c2 -1 0] after update
DS(4,9:12)=[0 0 0 -1]; % DS(4,9:12)= [0 0 0 c3] after update
```

```
%-----
%
% Define cost weighting parameters for optimization
```

```
    % state error weighting coefficients for cost function
```

```
w0=[1      0      0      0
     0      1      0      0
     0      0     0.1     0
     0      0      0     0.1];
```

```
    % process noise covariance
```

```
x0=[0      0      0      0
     0      1      0      0
     0      0     0.1     0
     0      0      0     0.1];
```

```
%-----
fprintf('          Load complete. \n\n');
return
```

```
% end of the bulddin function $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

### E.1.3 Check Dimensional Consistency: check.m

```
function consts=check(AB,DS,seqptr,theta,varptr,W,x0,BTP,keyseq);
%
fprintf('Checking dimensional consistency and computing [consts].\n');

% 'consts' is used by most other functions.
% 'consts' = [nx,nxc,nxs,nDS,nvar,nvarc,nseq,seqlen,keyseq,BTP,
%            fastseq,trainlen,period]
% where:
% 1. nx = total number of states
% 2. nxc = number of continuous states (A is nxc by nxc)
% 3. nxs = number of sample and hold states (B is nxc by nxs)
```

```

%      4. nDS = number of blocks in DS
%      5. nvar = number of variable parameters
%      6. nvarc = number of continuous (i.e. in AB) variable parameters
%      7. nseq = number of sample sequences
%      8. seqlen = length of longest sequence
%      9. keyseq = index of sequence synchronous with BTP
%     10. BTP   = basic time period
%     11. fastseq = index to shortest sequence
%     12. trainlen = length of shorter train
%     13. speriod = period of shortest sequence
%     Also: nxd = number of discrete states (nx = nxc + nxs + nxd)
%           ABwide = nxc+nxs
%           DSwide = nxs+nxd
%           phirow = block index to last row (phi's) in pc_phi array

% m is a dummy variable
consts=zeros(1,13);
[consts(1),m]=size(DS);
consts(4)=fix(m/consts(1));
    if rem(m,consts(1)), errormsg('ERROR, extra/missing DS columns.');
```

end;

```

[m,ABwide]=size(AB);
    if m~=ABwide, errormsg('DATA ERROR, AB matrix isnt square.');
```

end;

```

consts(2)=max(find(any(AB')));
consts(3)=ABwide-consts(2);
[consts(5),m]=size(varptr);
    if m~=3, errormsg('DATA ERROR, varptr does not have 3 columns.');
```

end;

```

    if consts(5)~=length(theta),errormsg('ERR, theta varptr mismatch.');
```

end;

```

consts(6)=sum((varptr(:,1)==0));
[consts(7),consts(8)]=size(seqptr);
    if consts(8)<2, errormsg('ERROR, longest sequence has 1 event.');
```

end;

```

        % check "varptr" consistency
for i=1:consts(5),
    if varptr(i,1)<0, errormsg('DATA ERROR, varptr(i,1) is negative.');
```

end;

```

    select=abs(varptr(i,1));
    if select>consts(4), errormsg('ERROR, varptr(i,1) > DS size.');
```

end;

```

    if rem(select,1)~=0, errormsg('ERROR, varptr(i,1) not integer.');
```

end;

```

    row=varptr(i,2); col=varptr(i,3);
    if row<1, errormsg('DATA ERROR, varptr(i,2) <=0.');
```

end;

```

    if col<1, errormsg('DATA ERROR, varptr(i,3) <=0.');
```

end;

```

    if select==0,
        if row>consts(2), errormsg('DATA, varptr(i,2) exceeds AB size.');
```

end;

```

        if col>ABwide, errormsg('DATA, varptr(i,3) exceeds AB size.');
```

end;

```

    else,
        if row>consts(1), errormsg('DATA, varptr(i,2) exceeds DS size.');
```

end;

```

        if col>consts(1)*consts(4), errormsg('varptr(i,3) exceeds DS.');
```

end;

```

    end; % end if select==0-else
end; % end for-loop
```

```

consts(9)=keyseq;           % sequence synched with BTP
consts(10)=BTP;             % BTP

% compute data for train calculations
etimes=seqptr.*(seqptr>0);   % elapsed time increments
periods=sum(etimes');       % periods of sequences (row)
[consts(13),consts(11)]=min(periods); % shortest period
etimes(consts(11),:)=zeros(1,consts(8)); % zero-out short sequence
bigno=max(max(etimes));     % largest element
etimes=etimes+bigno*(etimes==0); % set all the zeros to bigno
et=min(min(etimes));        % shortest etime in slower sequences
consts(12)=fix(min(et/consts(13))-1); % minimum cycles in train
consts(12)=max([1 consts(12)]); % avoid zero length train
consts(12)=consts(12)*consts(13); % minimum train length (time)

fprintf('          Check complete.\n\n');

% end of procedure check $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

```

#### E.1.4 Install Gains in System: update.m

```

function [AB, DS] = update(AB,DS,vars,varptr,consts);
%
% This function updates the AB and DS matrices to reflect the 'new'
% parameters in 'vars'. Also, if AB is changed, reset is set to one;
% otherwise it is zero.

nx=consts(1);           % total number of states
nvar=consts(5);         % number of variable parameters

for i=1:nvar,
    select=abs(varptr(i,1));
    row=varptr(i,2); col=varptr(i,3);
    if select==0,
        if AB(row,col)~=vars(i), AB(row,col)=vars(i); end;
    else,
        DS(row,((select-1)*nx+col))=vars(i);
    end; % end if-else
end; % end for-loop

% end of update function $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

```

#### E.1.5 Find Phase of $\Psi$ Discontinuities: cases.m

```

function [ctimes]=cases(flag,DS,seqptr,consts);

```

```

%
% This function computes the phase delay time boundaries which partition
% the sample schedule offset (tau) into regions of continuous Psi(tau).
% If flag=0 the continuity of the current Psi(k,tau) is considered. If
% flag=1, the continuity of Psi(k-1,tau) of the following BTP is
% considered too.
% 'ctimes'=[0] if nseq=1. ctimes=[0 tau-max] if otherwise synchronous.

% *****
% ** This version is limited to nseq=1 or 2. **
% *****

fprintf('Finding all continuous regions ... ');

nx=consts(1);          % total number of states
nseq=consts(7);         % number of sample sequences
seqlen=consts(8);       % length of longest sequence
keyseq=consts(9);       % Index to sequence synchronous with BTP
BTP=consts(10);         % basic time period
tol=1.0e-10;           % tolerance for same event times

varseq=3-keyseq;        % index to async sequence (keyseq=1 => varseq=2)

if (flag~=1)&(flag~=0), errmsg('flag out of range in cases.');
```

end;

```

if nseq==1; ctimes=[0]; return; end; % single sequence case
if nseq>2, errmsg('EXECUTION ERROR, More than 2 sequences.');
```

end;

```

if keyseq>2, errmsg('EXECUTION ERROR, KEYSEQ more than 2.');
```

end;

```

if keyseq<1, errmsg('EXECUTION ERROR, KEYSEQ less than 1.');
```

end;

```

etimes=(seqptr.*(seqptr>0));          % elapsed time intervals
period=sum(etimes);                   % periods of sequences

cycles(keyseq)=fix(BTP/period(keyseq));
    if rem(BTP,period(keyseq))~=0, errmsg('Key Sequence is Async.');
```

end;

```

cycles(varseq)=fix(BTP/period(varseq)); % max # full cycles in BTP
    seqrem=rem(BTP,period(varseq));    % remainder

    % cases with different numbers of full cycles in BTP
ctimes=[0];                          % starting points (min tau)
%if seqrem==0, return; end;           % all sequences synchronous
ctimes=[ctimes,seqrem];               % current BTP loses 1 cycle
if flag==1;                           % previous BTP gains one cycle
    temp=rem(2*seqrem,period(varseq));
    ctimes=[ctimes,temp];
end;
ctimes=[ctimes,period(varseq)];        % end point (max tau)

    % cases where discrete transitions don't commute
```

```

% Test all cases where:
%   i1*period(keyseq)+etimes(keyseq,j1) =
%   i2*period(varseq)+etimes(varseq,j2)+tau < BTP
%
% where: i1,i2,j1,j2 = non-negative integers & 0 < tau < period(varseq).

lim_kl=0;   lim_ku=fix((flag+1)*BTP/period(keyseq));
lim_vl=0;   lim_vu=fix((flag+1)*BTP/period(varseq));

for i1=-lim_kl:lim_ku,                                % full periods of keyseq
    tk=i1*period(keyseq);                               % time for full periods
    for j1=1:seqlen,                                   % part periods of keyseq
        if etimes(keyseq,j1)>=0,                       % test for continuous
            tk=tk+etimes(keyseq,j1);                   % event time
            if (tk>0)&(tk<=(flag+1)*BTP);              % event in BTP range
                for i2=lim_vl:lim_vu,                   % full varseq periods
                    tv=i2*period(varseq);               % time for full periods
                    for j2=1:seqlen,                     % part varseq periods
                        if etimes(varseq,j2)>=0,         % test for continuous
                            tv=tv+etimes(varseq,j2);    % event time w/o offset
                            if (tv>0)&(tv<tk);           % event in BTP & tau>0
                                if (tk-tv) < period(varseq), % if offset < period

                                    % check for discrete, non-commuting transitions
                                    spindx(keyseq)=j1; spindx(varseq)=j2;
                                    spindx(keyseq)=spindx(keyseq)+1;
                                    spindx(varseq)=spindx(varseq)+1;
                                    spindx=spindx-(spindx>seqlen)*seqlen;
                                    keytype=-seqptr(keyseq,spindx(keyseq));
                                    vartype=-seqptr(varseq,spindx(varseq));

                                    if keytype>0,          % synch event discrete
                                    if vartype>0,          % asynch event discrete
                                        temp1=DS(:,(1+(keytype-1)*nx):(keytype*nx))+...
                                                DS(:,(1+(vartype-1)*nx):(vartype*nx));
                                        temp2=DS(:,(1+(vartype-1)*nx):(vartype*nx))+...
                                                DS(:,(1+(keytype-1)*nx):(keytype*nx));
                                        if any(temp1-temp2),
                                            ctimes=[ctimes,(tk-tv)]; end; % non commuting
                                        end; end;
                                    % check complete, ctimes set if discrete & non-commuting

                                end; end; end; end; end; end; end; end; end; % end all if's & for's

% patch here to force an additional region(s)
% ctimes=[ctimes .35 .445 .55];
% end of special patch

```



```

ctimes=sort(ctimes);                % sort sequentially
ctimes=ctimes.*(ctimes>0);          % remove negative times
cases=length(ctimes)-1;
uniq=[1,((ctimes(2:cases+1)-ctimes(1:cases))>tol)];
ctimes=ctimes(uniq);                % eliminate duplicates

fprintf('\n')
% end of function cases $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

```

### E.1.6 Display Time and Date: shotime.m

```

function shotime(clock)

% Display time and date on screen in standard format

fprintf('\n');
fprintf(int2str(clock(4)));
fprintf(':');
fprintf(int2str(clock(5)));
fprintf(':');
fprintf(num2str(clock(6)));
fprintf(' ');

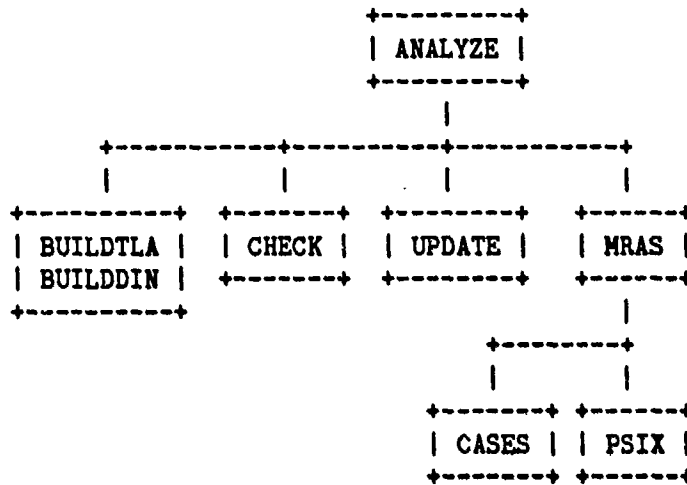
fprintf(int2str(clock(2)));
fprintf('/');
fprintf(int2str(clock(3)));
fprintf('/');
fprintf(int2str(clock(1)));
fprintf('\n');

% end of utility function shotime $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

```

## E.2 Analysis Routines

### E.2.1 Top-level Structure



### E.2.2 User Interface: analyze.m

% DRIVER SCRIPT FOR MULTIRATE ASYNCHRONOUS STABILITY ANALYSIS

```
shotime(clock);           % time back to estimate compute time
```

%

```
% This block creates the plant and cost function
% description matrices. Replace this block with code,
% input statements, and/or function calls that will create
% AB, DS,seqptr,theta,varptr,W,x0,BTP,and keyseq for
% the system you want to analyze.
```

## 2. LOAD PLANT & INITIAL CONTROLLER MODELS.

```

ssel=input('Select system: 1 => tla, 2 => dint: ');
if ssel==1,
    [AB,DS,seqptr,theta,varptr,W,x0,BTP,keyseq]=buildtla;
else,
    if ssel==2;
        [AB,DS,seqptr,theta,varptr,W,x0,BTP,keyseq]=bulddin;
    else,
        fprintf('\n Unknown system type...abort run. \n');
    end;
end;
end;

```

% SELECT INITIAL GAINS (theta)

```

tsel=input('Select initial theta; 1=>last run, 2=>loaded values: ');
if tsel==1, % load last saved theta vector
    if ssel==1, load tltheta.mat, else,
        if ssel==2, load dintheta.mat, end; end;
end;

%*****

% VERIFY DIMENSIONS AND COMPUTE [consts]
consts=check(AB,DS,seqptr,theta,varptr,W,x0,BTP,keyseq);
consts(15)=.5; % assume stable with initial gains

% INSTALL THETA VALUES IN AB AND DS MATRICIES
[AB, DS] = update(AB,DS,theta,varptr,consts);

% DISPLAY INITIAL GAINS
fprintf('\nTheta = %2.3g',theta(1));
for jp=2:consts(5) fprintf(' %0.3f',theta(jp)); end;
fprintf('\n\n');

% ***** CALL MULTI-RATE ASYNCH STABILITY EVAL ROUTINE *****

shotime(clock); % time hack to estimate compute time
fprintf('Compiling asynchronous stability test code: ..');

stability=mras(AB,DS,seqptr,consts);

fprintf(' The average stability figure is: %g',stability);

% The resulting stability figure is analogous to the real part
% of a continuous-time pole. "stability"<0 implies the system
% is stable with the error state decay rates bounded above
% (on average) by exp(stability*t).

% ***** END OF EVALUATION CALCULATIONS *****

shotime(clock); % time hack to estimate compute time

% end of main program $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

```

### E.2.3 Main Program: mras.m

```

function stability=mras_eval(AB,DS,seqptr,consts);

% Evaluates multi-rate asynchronous stability. 'stability' is
% analogous to an upper bound on the real part of the poles of
% a linear, time-invariant continuous-time system.

```

```

% *****
% ** This version is limited to nseq=2 **
% *****
fprintf(' Compile complete.\n');

% ***** SET CONSTANTS *****
nx=consts(1);          % number of states
nxc=consts(2);          % number of continuous states
nxs=consts(3);          % number of sample & hold states
nseq=consts(7);         % number of sequences
seqlen=consts(8);       % maximum sequence length
keyseq=consts(9);       % Index to sequence synchronous with BTP
BTP=consts(10);         % basic time period
fastseq=consts(11);     % fast sequence number
trainlen=consts(12);    % length of shorter train
speriod=consts(13);     % period of shortest sequence
varseq=3-keyseq;        % Index to sequence with variable phase
ABwide=nxc+nxs;
cycles=round(trainlen/speriod);

tol=0.01;              % tolerance for integral convergence
maxstep=.1*speriod;    % maximum integration step size
minstep=.01*speriod;   % minimum integration step size

% ***** SET ARRAY INDICES *****
cols=1:nx;              offcols=cols-nx;
phirows=1:nxc;          phicols=1:ABwide;
DSrows=nxc+1:nx;

% ***** COMPUTE TRAIN STM's *****
colsel=-seqptr(fastseq,1)*nx+offcols; % DS cols for end events
Psi=DS(:,colsel);        % initial Psi
for j=2:seqlen;
    isp=-seqptr(fastseq,j);          % event identifier
    if isp>0,                         % discrete transition
        Psi(DSrows,:)=DS(DSrows,offcols+isp*nx)*Psi;
    else, % isp must be < 0           % continuous time transition
        phi=expm(-AB*isp);           % continuous STM (isp=-t)
        Psi(phirows,:)=phi(phirows,:)*Psi(phicols,:);
    end; % if isp>0 ... else ...
end; % for j=2:seqlen                % no more discretes in speriod
trains=Psi*cycles;                   % Extend to 'cycles' periods
trains=[trains Psi*trains];          % Extend to 'cycles+1' periods
trains(DSrows,:)=DS(DSrows,colsel)*trains; % Final discrete event

% ***** INTEGRATIONS *****

```

```

brktimes=cases(1,DS,seqptr,consts);
nzones=length(brktimes)-1;
phasing=zeros(nseq,1);
varseq=3-keyseq; % index to slipping sequence
ets=seqptr(varseq,:); % pointer row for slipping sequence
vperiod=sum(ets.*(ets>0)); % period of slipping sequence
slip=rem(BTP,vperiod); % phase decrease in next BTP
plott=[]; plotSR=[]; plotSIG=[]; % clear plot data arrays
I_lnSR=0; I_lnSIG=0; stab=0; % initialize log integrals to zero
%fprintf('Slip=%f\n',slip);
% phasing(varseq)=NextTau+vperiod*((NextTau<0)-(NextTau>vperiod));

for zone=1:nzones; % step through zones
    tau=[]; LSR=[]; LSIG=[]; % null out data vectors
    ts=brktimes(zone);
    te=brktimes(zone+1);
    tau(1)=(te+ts)/2;
    fullspan=te-ts; % tau range of interval
    span=fullspan; % Euler integration step size
    oldpoints=0; % no. previous data
    newpoints=1; % start with 1 Euler point
    zstab=0; % initialize zone stability figure
    good=0; % assume integral not converged

    fprintf('\nZone Start Tau=%f, Zone End Tau=%f\n',ts,te);

    while good==0, % repeat until integral converges
        %fprintf('oldpoints=%f, newpoints=%f',oldpoints,newpoints);
        %fprintf('tau range=%g, integ step=%g.\n',fullspan,span);
        newindx=(oldpoints+1):newpoints; % indices to new points
        for i=newindx, % compute data at newpoints
            phasing(varseq)=tau(i); % phase for current BTP
            [ThisPL,ThisSR]=psix(phasing,AB,DS,seqptr,consts,trains,...
                                cols,offcols,phirows,phicols,DSrows,i);
            LSR(i)=log(ThisSR);
            NextTau=tau(i)-slip; % phase of next BTP
            phasing(varseq)=NextTau+vperiod*((NextTau<0)-(NextTau>vperiod));
            [NextLPI]=psix(phasing,AB,DS,seqptr,consts,trains,cols,...
                            offcols,phirows,phicols,DSrows,2);
            LSIG(i)=log(norm(NextLPI*ThisPL));
            %fprintf('step=%f, phasing=%g, LSR=%g',i,tau(i),LSR(i));
            %fprintf('LSIG=%g.\n',LSIG(i));
        end; % for i=newindx
        % add new data to plot vectors

        plott=[plott;tau(newindx)];
        plotSR=[plotSR;LSR(newindx)'];
        plotSIG=[plotSIG;LSIG(newindx)'];
    end
end

```

```

%   save mrasdat plott plotSR plotSIG   % comment out for clean output
shg
plot(plott,plotSR,'x',plott,plotSIG,'o');

    % compute/compare integrals
ILSR=span*sum(LSR);
ILSIG=span*sum(LSIG);
oldzstab=zstab;
zstab=ILSR+ILSIG;
error=abs( (zstab-oldzstab)/zstab );
%fprintf(' Integral ln(SR)=%g, Integral ln(SIG)=%g\n',ILSR,ILSIG);
%fprintf(' Zone Stab(N/O/C)=%g/%g/%g.\n',zstab,oldzstab,error);
    if span<minstep, good=-1; end;      % convergence failure
    if (error<tol)&(span<maxstep),
        good=1;
    else, % if (abs(... ))
        span=span/3;                    % cut integration step size by 3
        oldpoints=newpoints;            % remember old number of points
        newpoints=3*oldpoints;          % set new number of points
        tau=[tau;tau-span;tau+span];    % new phase time vector
    end; % if (abs(... )) else
end; % while good==0

I_lnSR=I_lnSR+ILSR;
I_lnSIG=I_lnSIG+ILSIG;
stab=stab+zstab;
fprintf('Zones 1-%.0f: I_ln(SR)= %.3g',zone,I_lnSR);
fprintf(' , I_ln(SIG) = %.3g, Stab= %.3g. \n',I_lnSIG,stab);
end; % for zone=1:nzones
save mrasdat plott plotSR plotSIG
stability=(stab/(brktimes(nzones+1)-brktimes(1)))/BTP;
return;

% end of function mras #####

```

## E.2.4 STM Calculation: psix.m

```

function [Pout,SR]=...
psix(t2go,AB,DS,sptr,con,trains,cols,offcols,phirows,phicols,DSrows,sflag);

% state transition matrix/factors for mras.m

t0=t2go;
nx=con(1);          nxc=con(2);          nxs=con(3);
nvar=con(5);        nvarc=con(6);         nseq=con(7);
seqlen=con(8);      BTP=con(10);          fastseq=con(11);
trainlen=con(12);   speriod=con(13);      ABwide=nxc+nxs;

```

```

%                                PART 1: COMPUTE STM FOR BASIC TIME PERIOD

%-----Set Pointers for Initial Discrete Events-----
%      t2go(i) will be the time from BTP start to the first discrete
%      event for each sequence. 'sptr(i,spindx(i))' defines the event.
done=1;  spindx=ones(t2go)*seqlen;  laststep=zeros(t2go);
while done>0;                                % completion flag
    for i=1:nseq,laststep(i)=sptr(i,spindx(i));end; % index to last event
    laststep=laststep.*(laststep>0);           % mask discrete events
    backstep=(laststep<t2go);                   % boolean
    done=sum(backstep);                         % 0 if done
    t2go=t2go-backstep.*laststep;              % adjust t2go
    t2go=t2go.*(t2go>0);                       % eliminate any negative times
    spindx=spindx-backstep;                     % adjust index
    spindx=spindx+seqlen*(spindx<1);           % fix possible wrap-around
end; % while done>0
spindx=spindx+1;                               % set to current event
spindx=spindx-(spindx>seqlen)*seqlen;         % fix wrap-around

%-----Compute Psi, the STM for specified t2go.-----

%                                INITIALIZE
zerotol=eps*BTP;
done=1;                                elaptime=0;           % time into BTP
[t1,i1]=min(t2go);                     % time to first event
isp=-sptr(i1,spindx(i1));               % identify first stm
Psi=eye(nx);                            % initialize Psi
%                                QUICKIE INITIAL DISCRETE EVENT IF PRACTICAL
if isp>0,                               % first event discrete
    blksel=isp*nx+offcols;
    Psi=DS(:,blksel);
    spindx(i1)=spindx(i1)+1;
end; % if isp>0

%                                MAIN STATE TRANSITION MATRIX (stm) CALCULATION LOOP
while done>0,                            % main Psi loop
    if (elaptime+t1+zerotol)>=BTP,t1=BTP-elaptime;done=0; end; % last step
    spindx=spindx-(spindx>seqlen)*seqlen; % fix any wrap-around
    if t1<=0,                            % t1=0 =>do next event now
        st=sort(t2go);                   % find next event time too
        t2=st(2);                        % time to second event
        if (i1==fastseq)&(t2>=trainlen)&(spindx(i1)==1)&...
            ((trainlen+elaptime)<BTP),    % next event is a train
            % TRAIN EVENT
            colsel=cols; dt=trainlen;    % use short train indices
            if (t2>=(dt+period))&((dt+elaptime+period)<BTP), % k cycles

```

```

        colsel=colsel+nx; dt=dt+speriod;      % use long train indices
    end;
    stm=trains(:,colsel);
    Psi=stm*Psi;
    t2go=t2go-dt;
    elaptime=elaptime+dt;
    t2go(i1)=t1;
else,
    isp=-sptr(i1,spindx(i1));                % next event NOT A TRAIN EVENT
    if isp>0,                                % event identifier
                                                % isp points to DS block
        % DISCRETE EVENT
        blksel=isp*nx+offcols;
        stm=DS(DSrows,blksel);
        Psi(DSrows,:)=stm*Psi;
    else, % isp must be < 0                    % i.e. continuous segment
        t2go(i1)=t2go(i1)-isp;                % increase that t2go element
    end; %if isp>0 ... else ...
end; % if (i1==fastseq ... else ...
    spindx(i1)=spindx(i1)+1;                  % increment sequence index
else, % if (t1<= 0)    i.e. t1>0 => time passes, continuous segment
    % CONTINUOUS TRANSITION
    phi=expm(AB*t1);
    Psi(phirows,:)=phi(phirows,:)*Psi(phicols,:);
    elaptime=elaptime+t1;
    t2go=t2go-t1;
end; % if t1 <=0 ... else ...
    [t1,i1]=min(t2go);                        % next event & its sequence
end; % while elaptime+t1<BTP                  % no more discretes in BTP

%*****End of Psi Calculation*****

%          PART 2: COMPUTE BALANCED FACTORS FOR STABILITY TEST
%          sflag==1 --> factors for "current" BPT (lambda + delta)
%          sflag==2 --> factors for "next" BTP    (delta only)

[P L]=eig(Psi);
PI=inv(P);
L=diag(L);
SR=norm(L,inf);
L=sqrt(L/SR);

for i=1:nx
    p=P(:,i);      np=norm(p);
    pinv=PI(i,:);  npi=norm(pinv);
    bal=sqrt(npi/np);
    if sflag==2, Pout(i,:)=pinv*L(i)/bal;
        else, Pout(:,i)=p*L(i)*bal; end;
end; % for i=1:nx

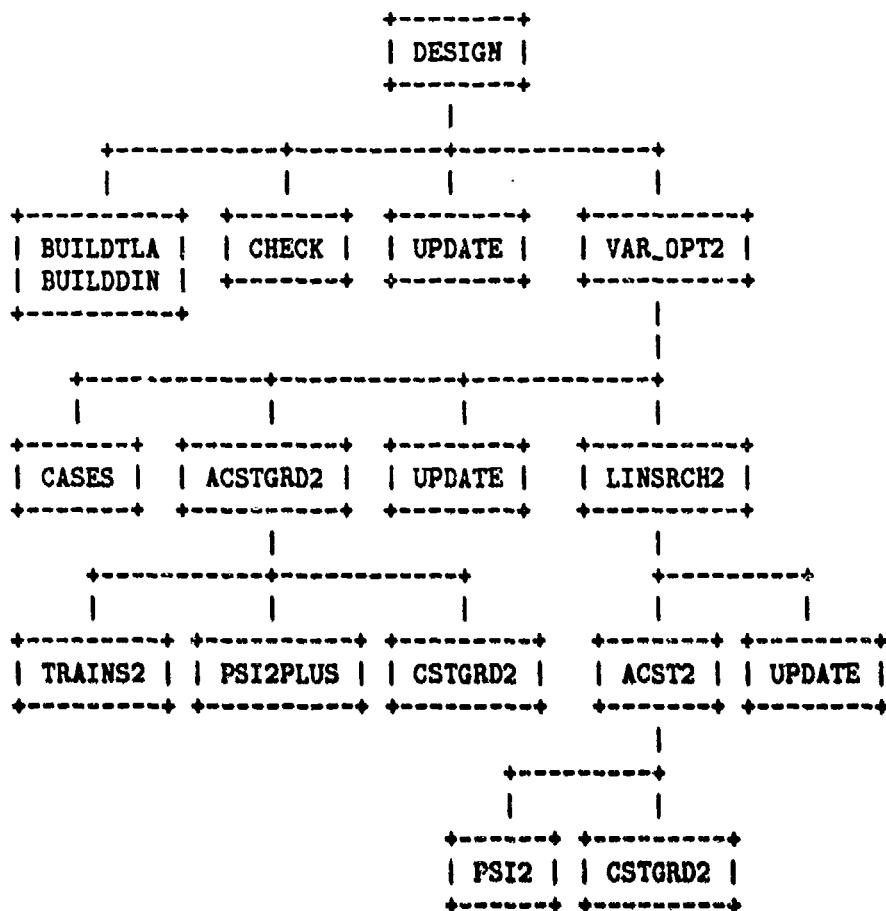
```





## E.3 Design Routines

### E.3.1 Top-level Structure



### E.3.2 User Interface: design.m

% Driver script for constrained asynchronous design algorithm

shotime(clock);

% time hack to estimate speed

% LOAD PLANT & INITIAL CONTROLLER MODELS.

ssel=input('Select system: 1 => tla, 2 => dint: ');

if ssel==1,

[AB,DS,seqptr,theta,varptr,W,x0,BTP,keyseq]=buildtla;

else,

if ssel==2;

[AB,DS,seqptr,theta,varptr,W,x0,BTP,keyseq]=bulddin;

```

else,
    fprintf('\n Unknown system type...abort run. \n');
end;
end;

% VERIFY DIMENSIONS AND COMPUTE [consts]
consts=check(AB,DS,seqptr,theta,varptr,W,x0,BTP,keyseq);
consts(15)=.5; % assume system is stable with initial gains

% SELECT & INSTALL INITIAL GAINS (theta)
tsel=input('Select initial theta; 0=>random, 1=>last run, 2=>loaded: ');
if tsel==0,
    consts(15)=2; % assume unstable if rand gains
    theta=rand(theta)-.5; % random gains: U(-0.5, 0.5)
end;
if tsel==1, % load saved theta vector (assume stable)
    if ssel==1, load tlatheta.mat, else,
    if ssel==2, load dintheta.mat, end;end;
end;
[AB, DS] = update(AB,DS,theta,varptr,consts);

% DISPLAY INITIAL GAINS
fprintf('\nTheta = %2.3g',theta(1));
for jp=2:consts(5) fprintf(' %0.3f',theta(jp)); end;
fprintf('\n');

% ***** CALL DESIGN PROGRAM, OPTIMIZE [theta] *****

% Select synchronous/asynchronous phasing
consts(14)=input('Select method; 0=>Asynch, 1=>Synch: ');
consts(15)=...
    input('Select initial cost function; 0=>Stable, 1=>Unstable: ');
shotime(clock); % time hack to estimate speed

fprintf('Compiling parameter optimization code:'); % user advisory
[theta,AB,DS]= var_opt2(AB,DS,W,x0,theta,varptr,seqptr,consts,ssel);

shotime(clock); % time hack to estimate speed

% end of main program listing $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

```

### E.3.3 Main Optimization Program: var\_opt2.m

```

function [theta,AB,DS]=...
    var_opt(AB,DS,w0,x0,theta,varptr, seqptr,consts,ssel);

% main driver for coefficient optimization

```

```

fprintf(' Optimization compile complete.\n')
fprintf('\nBegin parameter optimization (var_opt). \n')

% for single-sequence or synchronous cases: times=[0]
% for the two-sequence asynchronous cases: [times] partitions the
% time offset (phase) into regions of continuous 'Psi'.

nx=consts(1);           % number of states
nvar=consts(5);         % number of variable parameters
seqlen=consts(8);       % length of longest sequence
fastseq=consts(11);     % index to shortest sequence
oldalpha=1e-6;          % some guess to initialize step size
grad=ones(1,nvar);      % allocate space
srchdir=ones(nvar,1);   % allocate space
alpha=0;                % allocate space

complete=0;             % completion flag, +1 if ok, -1 if timeout
loopcount=1;            % counter to initialize Hessian
totalcount=1;           % total iteration counter
maxcount=50;            % max iteration limit (timeout trigger)
tol=.0001;              % gradient convergence criteria
mincost=tol;            % cost convergence criteria
nomstep=.001;           % nominal alpha guess for first search

% PARTITION OF CONTINUOUS PHASE REGIONS
[times] = cases(0,DS,seqptr,consts);

% FIND CONTINUOUS AND DISCRETE VARIABLES IN FASTSEQ
cvarsel=(varptr(:,1)==0); % boolean of cont. vars.
tvarsel=cvarsel;          % boolean of all vars
for i=1:seqlen,           % discrete vars. in fastseq
    tvarsel=tvarsel|(varptr(:,i)==-seqptr(fastseq,i));
end;
block=cumsum(tvarsel).*tvarsel-1; % block offset for partials
block=nx*block;           % column offset for partials
tvarsel=find(tvarsel)';   % all vars. in fastseq
cvarsel=find(cvarsel)';   % continuous vars (in AB)

% ***** MAIN SEARCH LOOP *****
while complete==0,
    fprintf('\nStart gradient search number ');
    fprintf(int2str(totalcount)); fprintf('\n');
    lastgrad=grad;
    [cost,grad,stable]=...
        acstgrd2(times,w0,x0,AB,DS,seqptr,varptr,consts,block,tvarsel,cvarsel);
    delgrad=grad-lastgrad;
    gradnorm=norm(grad);

```

```

% COMPUTE SEARCH DIRECTION
% quasi-Newton search using Broyden-Fletcher-Goldfarb-Shanno update.

% update Hessian, "G"
if loopcount==1, % reset Hessian to I
    G=eye(nvar); nsd=1; oldalpha=nomstep;
else, % if loopcount==1 % compute Hessian update
    G=G+lastgrad'*lastgrad/(lastgrad*srchdir)+...
        delgrad'*delgrad/(alpha*delgrad*srchdir);
end; % if loopcount==1,
if min(real(eig(G)))<=eps, G=eye(nvar); end; % insure G positive def.

% compute search direction & step estimate
oldalpha=oldalpha*nsd;
srchdir=-G\grad'; % Newtonian search direction
nsd=norm(srchdir);
oldalpha=oldalpha/nsd; % initial step size estimate
fprintf('Cost=%6.5g. ||Grad||=%6.5g. ||SrchDir||=%6.5g\n',cost,gradnorm,nsd)

% COMPUTE STEP (ALPHA) TO MIN COST

alpha=linsrch2(srchdir,oldalpha,cost,times,w0,x0,AB,DS,theta,...
    seqptr,varptr,consts);
if alpha<=eps, alpha=0; loopcount=0; end;
oldalpha=alpha+1e-6; % ensure old step is positive

% UPDATE PARAMETER SET

theta=theta+alpha*srchdir; % update theta
[AB,DS]=update(AB,DS,theta,varptr,consts); % update AB and DS

if ((stable-1)*(consts(15)-1))<0, loopcount=1; end;
consts(15)=stable;

fprintf('\nTheta = %2.3g',theta(1));
for jp=2:nvar fprintf(' %2.3g',theta(jp)); end;
fprintf('\n');

% SAVE GAINS FOR FUTURE REFERENCE

if ssel==1,
    save tlatheta.mat theta;
else,
    if ssel==2, save dintheta.mat theta; end;
end;

% TEST FOR COMPLETION/TIMEOUT

```

```

if totalcount>maxcount, complete=-1; end;
if gradnorm<tol*cost, complete=1; end;
if cost<mincost, complete=1; end;
% if loopcount<3*nvar, loopcount=0; end;      % arbitrary Hessian reset
loopcount=loopcount+1;
totalcount=totalcount+1;
end; % while complete~= (main search loop)

return;
% end of function min_var $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

```

### E.3.4 Asynchronous Cost and Gradient: acstgrd2.m

```

function [cost,grad,stable]=...
acstgrd2(brktimes,w0,x0,AB,DS,seqptr,varptr,consts,block,tvarsel,cvsel);

% estimate average cost & gradient over phase using polynomial curve fit

xsafe=.99;          % end point setback insures unique transition

% *****
% ** This version is limited to nseq=1 or 2 only. **
% *****

%*****
%
% OUTPUTS: cost = phasing-weighted asynchronous cost function (scalar)
%          grad = gradient of cost WRT theta (column vector)
%          stable = flag to select correct cost function in linrch.m
% INPUTS: brktimes = sorted vector of tau's that partition regions of
%          continuous Psi(tau).
%          w0,x0,AB,DS,seqptr,varptr,consts. (see build & check)
%
%*****

%***** STATE TRANSITION MATRIX PARAMETERS*****
%
% These parameters are defined for any time period:
% Psi = State Transition Matrix for the period
% PsiDot = partial of Psi wrt a variable parameter (theta element)
% W = weighting:  $x(0)'W x(0) = \int_{\text{period}} x(t)'W x(t) dt$ 
% W_dot = partial W wrt a variable parameter (theta element)
% X =  $E\{x(f)x(f)'\}$  from process noise (x0) during period (if x(0)=0)
% X_dot = partial of X wrt a variable parameter (theta element)
% Y =  $E\{\int_{\text{period}} x(t)'W x(t) dt\}$  if x(0)=0. (scalar)
% Y_dot = partial Y wrt a variable parameter (theta element)
%

```

```

% RECURSIVE ALGORITHM FOR COMPUTING STM PARAMETERS
% Suppose:  $\Psi(t_f, t_0) = T(n) * T(n-1) * \dots * T(2) * T(1)$ , where  $T(i)$  is
% the state transition matrix (stm) for the  $i$ 'th segment of the BTP.
% Let  $Q(i)$ ,  $R(i)$ , and  $S(i)$  represent  $W$ ,  $X$ , and  $Y$ , respectively, for  $T(i)$ .
%
% Initialize:
%    $\Psi(0) = I$ ,            $\Psi_{dot}(0) = [0, 0, \dots, 0, 0]$  These may be considered
%    $W(0) = 0$ ;            $W_{dot}(0) = [0, 0, \dots, 0, 0]$  correct values for  $T_i = I$ ,
%    $X(0) = 0$ ;            $X_{dot}(0) = [0, 0, \dots, 0, 0]$  a null transition with
%    $Y(0) = 0$ ;            $Y_{dot}(0) = [0, 0, \dots, 0, 0]$  zero elapsed time.
%
% Recursions:
%    $\Psi(i) = T(i) * \Psi(i-1)$ ,
%    $\Psi_{dot}(i) = T(i) * \Psi_{dot}(i-1) + T_{dot}(i) * \Psi(i-1)$ ,
%    $W(i) = W(i-1) + \Psi(i-1)' * Q(i) * \Psi(i-1)$ 
%    $W_{dot}(i) = W_{dot}(i-1) + \Psi_{dot}(i-1)' * Q(i) * \Psi(i-1) + \Psi(i-1)' * Q_{dot}(i) * \Psi(i-1) + \Psi(i-1)' * Q(i) * \Psi_{dot}(i-1)$ 
%    $X(i) = T(i) * X(i-1) * T(i)' + R(i)$ 
%    $X_{dot}(i) = T_{dot}(i) * X(i-1) * T(i)' + T(i) * X_{dot}(i-1) * T(i)' + T(i) * X(i-1) * T_{dot}(i)' + R_{dot}(i)$ 
%    $Y(i) = Y(i-1) + X(i-1) * Q(i)$ 
%    $Y_{dot}(i) = Y_{dot}(i-1) + X_{dot}(i-1) * Q(i) + X(i-1) * Q_{dot}(i)$ 
%
% Then,  $\Psi(n)$ ,  $\Psi_{dot}(n)$ ,  $W(n)$ ,  $W_{dot}(n)$ ,  $X(n)$ ,  $X_{dot}(n)$ ,
%  $Y(n)$  and  $Y_{dot}(n)$  apply to the period covered by  $\Psi$ .
%
% *****
% ***** TRAIN DATA *****
%
% Train concept: The fastest sampler has frequent bursts or trains of
%  $k$  or  $k-1$  consecutive short sample periods where:
%    $k = \text{trunc}(2 \times \text{nd shortest sample period} / \text{shortest sample period})$ .
% Data for these fixed STM's are precalculated and stored in 'trndat'
% to avoid subsequent reduce redundant calculations. 'trndat'
% includes the discrete events on both ends of the fast 'sample train.'
%
% The precalculated trndat and cc parameters are defined as:
%
%   QQ      =      [ W1 ; W2 ]
%   RR      =      [ X1 ; X2 ]
%   SS      =      [ Y1 ; Y2 ]
%   TT      =      [ Psi1 ; Psi2 ]
%
%   QQdot   =      [ W1_dot_1 ... W1_dot_nvart ;
%                   W2_dot_1 ... W2_dot_nvart ]
%   RRdot   =      [ X1_dot_1 ... X1_dot_nvart ;
%                   X2_dot_1 ... X2_dot_nvart ]

```

```

%      SSdot =      [ Y1_dot_1   ...   Y1_dot_nvar ;
%                    Y1_dot_2   ...   Y2_dot_nvar ]
%      TT      =      [ Psi1_dot_1   ...   Psi1_dot_nvar ;
%                      Psi2_dot_1   ...   Psi2_dot_nvar ]
%
% Where Psi1, W1, X1 and Y1 apply to the shorter train (k-1 periods),
% and Psi2, W2, X2, and Y2 apply to the longer train (k periods).
% The '_dot_i' notation indicates the partial with respect to the i'th
% variable (theta element) in the fast sequence (nvar such variables).
%
%*****

nx=consts(1);      nxc=consts(2);      nxs=consts(3);
nvar=consts(5);     nseq=consts(7);     seqlen=consts(8);
keyseq=consts(9);   varseq=3-keyseq;     fastseq=consts(11);
% note: synch = consts(14), synch==1 forces synchronous phasing
% note: stable = consts(15), stable>1 selects unstable cost/grad

if (nseq<1)|(nseq>2), errmsg('nseq out of range, acsgrd'); return; end;

% PRECOMPUTE DATA FOR FAST SAMPLE TRAIN
[QQ,RR,SS,TT,QQdot,RRdot,SSdot,TTdot]=...
    trains2(w0,x0,AB,DS,seqptr,varptr,consts,block,tvarsel,cvarsel);

stable=0;
synch = consts(14); % =1 for synch treatment
ncases=length(brktimes)-1;
if synch==1, ncases=0; end; % select synch/asynch
phasing=zeros(2,1);

if ncases==0, % this is a synchronous problem

% patch here to hard-wire synchronous phase (patch acst2 likewise)
% phasing(1)=eps;
% end of patch

fprintf(' Synch Cost & Gradient (one case).\n ');
[Psi,xxt,wyt,yyt,Lam,P,xxtdot,wtdot,yytdot,LamDot,PDot]=...
    psi2plus(phasing,AB,DS,seqptr,varptr,consts,x0,w0,QQ,RR,SS,TT,...
        block,tvar=1,QQdot,RRdot,SSdot,TTdot);
stable=max(abs(Lam));
[cost,grad]=cstgrd2(wyt,xxt,yyt,Psi,consts,Lam,P,wtdot,xxtdot,...
    yytdot,LamDot,PDot);

return;
else, % if ncases==0
    if nseq~=2, errmsg('EXECUTION ERROR, nseq~=2 in acsgrd'); end;
end;

```



```

fprintf(' Asynch Cost/Grad: %2.0f',ncases);
fprintf(' continuous phase regions. \n');

% compute percentages for each interval
tweights=brktimes(2:(ncases+1))-brktimes(1:ncases);
tweights=tweights/(brktimes(ncases+1)-brktimes(1));

cost=0;          grad=zeros(1,nvar);
t=zeros(3,1);    y=zeros(3,1);    g=zeros(3,nvar);
xa=1/xsafe;      xc=xa*xa*xa/3;

% COMPUTE \INTEGRAL_{PHASING} COST, AND GRADIENT FOR REGIONS
for i=1:ncases,
    ts=brktimes(i); te=brktimes(i+1); span=(te-ts)/2;
    t(2)=(ts+te)/2; t(1)=t(2)-xsafe*span; t(3)=t(2)+xsafe*span;

    for j=1:3; % get costs/grads for 3-point fit
        phasing(varseq)=t(j);
        [Psi,xxt,wt,yyt,Lam,P,xtdot,wtdot,yydot,LamDot,PDot]=...
            psi2plus(phasing,AB,DS,seqptr,varptr,consts,x0,w0,QQ,RR,SS,TT,...
                block,tvarsel,QQdot,RRdot,SSdot,TTdot);
        stable=max([abs(Lam); stable]);
        [y(j),g(j,:)] = cstgrd2(wt,xxt,yyt,Psi,consts,Lam,P,wtdot,xtdot,...
            yydot,LamDot,PDot);
    end; % for j=1:3

% approximate curve as:  $y = c x^2 + b x + a$ ,  $x \in (-1.01, 1.01)$ .
% where  $a=y(2)$ ;  $b=(y(3)-y(1))/2$ ;  $c=(y(3)-2*y(2)+y(1))/2$ .
% Then  $\int y(\tau) d\tau = c(2x^3)/3 + a(2x)$  with  $x=1/xsafe$ .

a=y(2);
adot=g(2,:);
c=(y(3)-2*y(2)+y(1))/2;
cdot=(g(3,:)-2*g(2,:)+g(1,:))/2;

% include current phase region in composite cost/grad
delcost=tweights(i)*(c*xc+a*xa);
cost=cost+delcost;
grad=grad+tweights(i)*(cdot*xc+adot*xa);

fprintf(' Region '); fprintf(int2str(i));
fprintf(' (tau=%6.5f-%6.5f).',ts,te)
fprintf(' DelCost=%6.5f. Cost=%6.5f.\n',delcost,cost)
end; % for i=1:ncases

return
% end of function acstgrd2 #####

```

## E.3.5 Asynchronous Cost Only: acst2.m

```

function cost=acst2(brktimes,w0,x0,AB,DS,seqptr,consts);

% estimate cost averaged over phase using polynomial curve fit
% see function acstgrd2 for detailed comments

% For validation testing of trndat & cc, declare as:
% function [QQ,RR,SS,TT]=trnstst2(w0,x0,AB,DS,seqptr,consts);
% and delete indicated portion at end.

% *****
% ** This version is limited to nseq=1 or 2 only. **
% *****

nx=consts(1);          nxc=consts(2);          nxs=consts(3);
nseq=consts(7);        seqlen=consts(8);        keyseq=consts(9);
fastseq=consts(11);    ABwide=nxc+nx;
cycles=round(consts(12)/consts(13));            % trainlen/speriod

if (nseq<1)|(nseq>2), errmsg('nseq out of range, acstgrd'); return; end;

%          fixed indices for array operations
cols=1:nx;              cols2=cols+nxc;          offcols=cols-nx;
phirows=1:nxc;          phi2rows=phirows+nxc;
phicols=1:ABwide;       phi2cols=phicols+ABwide;

%          COMPUTE PARAMETERS FOR ONE CYCLE OF FASTSEQ.

%          Put first event in train data
isp=-seqptr(fastseq,1);          % initial event number
blk1=isp+nx+offcols;
rowssel=find(any(DS(:,blk1)~-eye(nx)));          % non-identity rows
Q=zeros(nx);
R=zeros(nx);
S=0;
T=DS(:,blk1);

%          Include rest of first cycle in train data
for j=2:seqlen;
    isp=-seqptr(fastseq,j);          % event identifier
    if isp>0,                          % isp points to DS block
        %          DISCRETE EVENT
        blk=isp+nx+offcols;          % index to DS
        rowssel=find(any(DS(:,blk)~-eye(nx)));          % non-identity rows
        stm=DS(rowssel,blk);          % new stm
        R(rowssel,:)=stm*R;          % X=stm*X
        T(rowssel,:)=stm*T;          % Psi=stm*Psi
    end
end

```

```

R(:,rowssel)=R*stm'; % X=[stm*X]*stm'+process noise
R(rowssel,rowssel)=R(rowssel,rowssel)+x0(rowssel,rowssel);
else, % isp must be < 0 % continuous time only
% CONTINUOUS TIME TRANSITION
t1=-isp; ABt1=AB*t1; At1=ABt1(phirows,phirows);
tempwq=[-ABt1',w0(phicols,phicols)*t1;zeros(ABwide),ABt1];
tempxr=[-At1,x0(phirows,phirows)*t1;zeros(nxc),At1'];
tempwq=expm(tempwq);
tempxr=expm(tempxr);
phix=tempwq(phi2cols,phi2cols); % phi extended
phi=phix(phirows,:); % phi=[Phi(t1),Gamma(t1)];
Qt1=w0*t1; % Qt1=\integral_{0}^{-t1} B'exp(A't) w0 exp(At) B dt
Qt1(phicols,phicols)=phix'*tempwq(phicols,phi2cols);
% Rt1=\integral_{0}^{-t1} exp(At) x0 exp(A't) dt
Rt1=phi(:,phirows)*tempxr(phirows,phi2rows);
S=S+sum(sum(R.*Qt1)); % delta cost correct
qtemp=T'*Qt1; % Psi'*Q
R(phirows,:)=phi*R(phicols,:); % X=stm*X
Q=Q+qtemp*T; % W=W+stm'*Q*stm
T(phirows,:)=phi*T(phicols,:); % Psi=stm*Psi
R(:,phirows)=R(:,phicols)*phi'; % X=[stm*X]*stm'
R(phirows,phirows)=R(phirows,phirows)+Rt1; % X=[stm*X*stm']*R
end; % if isp>0 ... else ...
end; % for j=2:seqlen % no more discretizes in speriod

% EXTEND TO 'CYCLES' & 'CYCLES+1' PERIODS AND ADD FINAL DISCRETE EVENT
Psi=T;
X=R;
W=Q;
cc=S;
stm=DS(rowssel1,blk1); % stm for final event

loopcount=[cycles,2]; % set number of loop iterations
for case=1:2; % 1: first cycles periods; 2: last period
for i=2:loopcount(case),
cc=cc+sum(sum(X.*Q))+S;
qtemp=Psi'*Q;
X=T*X;
W=W+qtemp*Psi;
Psi=T*Psi;
X=X*T'+R;
end;
base=(case-1)*nx; colsel=base+cols; rowssel=base+rowssel1;
QQ(colsel,:)=W;
SS(case)=cc;
RR(colsel,:)=X;
RR(rowssel,:)=stm*X;
TT(colsel,:)=Psi; TT(rowssel,:)=stm*Psi;

```

```

RR(colsel,rowsel1)=RR(colsel,:)*stm';
RR(rowsel,rowsel1)=RR(rowsel,rowsel1)+x0(rowsel1,rowsel1);
end; % for case=1:2

% *****
%      DELETE EVERYTHING FROM HERE DOWN TO CREATE TRNTST.M
% *****

synch = consts(14);                % =1 for synch
ncases=length(brktimes)-1;
if synch==1, ncases=0; end;        % select synchronous
phasing=zeros(2,1);

if ncases==0,                      % this is a synchronous problem

% patch here to hard-wire synchronous phase (patch acstgrd2 likewise)
% phasing(1)=eps;
% end of patch

[Psi,xxt,wwt,yyt]=...
    psi2(phasing,AB,DS,seqptr,consts,x0,w0,QQ,RR,SS,TT);
cost=cstgrd2(wwt,xxt,yyt,Psi,consts);
return;
else, % if ncases==0
    if nseq~=2, errmsg('EXECUTION ERROR, nseq~=2 in acsrgrd'); end;
end;

y=zeros(3,1);
varseq=3-keyseq;
% compute percentages for each interval
tweights=brktimes(2:(ncases+1))-brktimes(1:ncases);
tweights=tweights/(brktimes(ncases+1)-brktimes(1));
cost=0; t=zeros(3,1); y=zeros(3,1);
xsafe=.99; % back-off from ends to ensure unambiguous transition
xa=1/xsafe; xc=xa*xa*xa/3;

% compute \integral_{phasing} cost, and gradient for regions
for i=1:ncases,

    ts=brktimes(i); te=brktimes(i+1); span=(te-ts)/2;
    t(2)=(ts+te)/2; t(1)=t(2)-xsafe*span; t(3)=t(2)+xsafe*span;

    for j=1:3; % get costs/grads for 3-point fit
        phasing(varseq)=t(j);
        [Psi,xxt,wwt,yyt]=...
            psi2(phasing,AB,DS,seqptr,consts,x0,w0,QQ,RR,SS,TT);
        y(j)=cstgrd2(wwt,xxt,yyt,Psi,consts);
    end; % for j=1:3

```

### E.3.6 Cost and Gradient Definition: cstgrd2.m

```
% This function computes the cost or cost plus gradient for the
% BTP state transition matrix Psi (i.e. at one phase condition).
% For cost alone, the last four input arguments are optional.
```

```
% One of two cost functions are selected by consts(15).
```

```
% If consts(15)>1,
%     cost=sum(abs(eig((Psi))).^8), 4'th order version commented out
%
% If consts(15)<1,
%     cost = xrt.*(P'\[1./((1-conj(Lam)*Lam.-1))-1).*(P'*wt.*P)].*/P)
```

```

%
%      This is the weighted (by wwt) value of the expected
%      noise power for specified process noise (xxt).
%

nx=consts(1);          % total number of states
nvar=consts(5);        % number of variable parameters
stable=consts(15);
BTP=consts(10);

if nargin<7,            % supply any missing arguments
    [P,Lam]=eig(Psi);
    Lam=diag(Lam);
    for i=1:nx, P(:,i)=P(:,i)/norm(P(:,i)); end;
end; % if nargin<5,

if stable>=1;           % use unstable cost function
    omega=conj(Lam).*Lam;
    temp=omega.^3;      % eighth order cost
    % temp=omega;        % fourth order cost
    cost=omega'*temp;
    if nargout>1,       % compute gradient too
        omega=temp;
        gradient=8*real((omega.*Lam)'.*LamDot);
    end;
else, % if stable>=1 ... else ... % use stable cost function
    ALam=abs(Lam);
    if max(ALam)>1-eps,   % if actually unstable
        cost=1/eps;     % near infinite cost
    else, % if max(ALam)... else ... % i.e. if stable
        omega=(1)./(1-conj(Lam).*Lam. ');
        wp=wwt*P;
        pwp=P'*wp;
        temp=omega.*pwp;
        invp=inv(P);
        temp1=real(invp'*temp*invp);
        cost=sum(sum(xxt.*temp1))+yyt;
        cost=cost/BTP;
        if nargout>1,   % compute gradient too
            omegasq=omega.*omega;
            cols=1-nx:0;
            for i=1:nvar, % for each variable
                inx=i+nx;
                pwpdot=PDot(:,inx+cols)'.*wp; % interrim result
                pwpdot=pwpdot+pwpdot'+P'*wwtdot(:,inx+cols)*P;
                omegdot=conj(Lam).*LamDot(:,i)'; % interrim result
                omegdot=(omegdot+omegdot')*.omegasq;
                tempdot=omega.*pwpdot+omegdot.*pwp;
            end;
        end;
    end;
end;

```

```

        temp3=(tempdot-2*temp*invp*PDot(:,inx+cols)); % interrim result
        temp2=xt.*real(invp'*temp3*invp)+xtdot(:,inx+cols).*temp1;
        gradient(i)=sum(sum(temp2))+yytdot(i);
    end; % for i=1:nvar
    gradient=real(gradient)/BTP;
end; % if nargout>1
end; % if max(AbsLam)>1-eps ... else ...
end; % if stable
cost=real(cost);
return;

```

```
% end of function cstgrd2 #####
```

### E.3.7 STM and Partial: psi2plus.m

```
function [Psi,XX,WW,YY,Lam,P,PXX,PWW,PYY,PLam,PP]=...
psi2plus(t2go,AB,DS,sptr,vptr,con,x0,w0,qq,rr,ss,tt,block,tvar,pqq,prp,pss,pt
```

```
% "psi2plus" computes BTP STM, related matrices, and gradients.
% See extensive comments at end of code.
```

```
fprintf('    psi2plus: ');
```

```

%          UNPACK CONSTANTS
nx=con(1);          nxc=con(2);          nxs=con(3);
nvar=con(5);          nvarc=con(6);          nseq=con(7);
seqlen=con(8);          BTP=con(10);          fastseq=con(11);
trainlen=con(12);          speriod=con(13);          ABwide=nxc+nxs;
%          CREATE INDICES FOR ARRAY OPERATIONS
cols=1:nx;          offcols=cols-nx;          varindx=1:nvar;
phirows=1:nxc;          x2cols=phirows+nxc;          x3cols=x2cols+nxc;
phicols=1:ABwide;          w2cols=phicols+ABwide;          w3cols=w2cols+ABwide;
offpcols=phicols-nx;          offprows=phirows-nx;
%          SET INDICES FOR PARTIALS CALCULATIONS
if nargout>6, nvar=length(tvar); cvarsel=find((vptr(:,1))==0)';
else, varindx=[]; tvar=[]; nvar=0; cvarsel=[]; nvarc=0; vptr(1,:)= -1; end;
%          VERIFY t2go IN RANGE
if length(t2go)~=nseq, errmsg('Extra/missing phase times. '); end;
if (any(t2go<0)), errmsg('Negative time to go in psi plus. '); end;

%-----Set Pointers for Initial Discrete Events-----
%          t2go(i) will be the time from BTP start to the first discrete
%          event for each sequence. 'sptr(i,spindx(i))' defines the event.
done=1;  spindx=ones(t2go)*seqlen;  laststep=zeros(t2go);
while done>0;          % completion flag
    for i=1:nseq, laststep(i)=sptr(i,spindx(i)); end; % index to last event
    laststep=laststep.*(laststep>0);          % mask discrete events

```

```

backstep=(laststep<t2go);           % boolean
done=sum(backstep);                 % 0 if done
t2go=t2go-backstep.*laststep;      % adjust t2go
t2go=t2go.*(t2go>0);                % eliminate any negative times
spindx=spindx-backstep;             % adjust index
spindx=spindx+seqlen*(spindx<1);    % fix possible wrap-around
end; % while done>0
spindx=spindx+1;                    % set to current event
spindx=spindx-(spindx>seqlen)*seqlen; % fix wrap-around

%-----Compute Psi, PsiDot, XX, PXX, WW, PW, YY, PYY-----
% Psi = psi, the state transition matrix for the specified t2go.
% PsiDot = [d Psi/d theta-1, ... , d Psi/d theta-nvar] {nxn block row}
% XX = E xx' at BTP end from process noise during BTP
% PXX = partials XX wrt theta {row of nxn blocks}
% WW = matrix s.t. x0'*WW*x0 is the cost for x0 error at BTP start
% PW = partials WW wrt theta {row of nxn blocks}
% YY = cost from process noise during BPT
% PYY = partials YY wrt theta {row of scalars}
% All used as accumulators for intermediate results during recursions.

% DECLARE VARIABLES SO STUFF TO BE CLEARED WILL BE AT END
Psi=eye(nx); PsiDot=zeros(nx,nvar*nx);
XX=zeros(nx); PXX=zeros(nx,nvar*nx);
YY=0; PYY=zeros(nvar,1);
WW=zeros(nx); PW=zeros(nx,nvar*nx);
i=0;j=0;inx=0;jnx=0;ir=0;iro=0;ic=0;ico=0;ispnx=0;
t2=0;offset=0;dt=0;st=zeros(t2go);ssel=0;

% INITIALIZE FOR FIRST EVENT
done=1; elaptime=0; % time into BTP
zerotol=BTP*eps;
[t1,i1]=min(t2go); % time to first event
isp=-sptr(i1,spindx(i1)); % identify first stm
% QUICKIE INITIAL DISCRETE EVENT IF PRACTICAL
if isp>0, % first event discrete
    blkssel=isp*nx+offset;
    Psi=DS(:,blkssel);
    varsel=find((vptr(:,1)==isp)');
    for i=varsel, PsiDot(vptr(i,2),(i-1)*nx+vptr(i,3))=1; end;
    spindx(i1)=spindx(i1)+1;
    fprintf('D'); fprintf(int2str(isp));
end; % if isp>0

% MAIN STATE TRANSITION MATRIX (stm) CALCULATION LOOP
while done>0, % main loop for Psi calcs
    if (elaptime+t1+zerotol)>=BTP,t1=BTP-elaptime;done=0; end; % last step
    spindx=spindx-(spindx>seqlen)*seqlen; % fix possible wrap-around

```



```

if t1<=0,                                     % t1=0 => do next event now
    st=sort(t2go);                             % find next event time too
    t2=st(2);                                   % time to second event
    if (i1==fastseq)&(t2>=trainlen)&(spindx(i1)==1)&...
        ((trainlen+elaptime)<BTP),             % next event is a train
        %
        TRAIN EVENT
        rowsel=cols; ssel=1; dt=trainlen;      % use short train indices
        if (t2>=(dt+period))&((dt+elaptime+period)<BTP), % k cycles
            rowsel=rowsel+nx; ssel=2; dt=dt+period;% use long train indices
        end;
        stm=tt(rowsel,:);
        YY=YY+sum(sum(XX.*qq(rowsel,:)))+ss(ssel);
        qtemp=Psi'*qq(rowsel,:);
        for i=varindx, blkssel=i*nx+offcols;    % for all variables
            qqdot=qtemp*PsiDot(:,blkssel);
            PW(:,blkssel)=PW(:,blkssel)+qqdot*qqdot';
            PYY(i)=PYY(i)+sum(sum(PXX(:,blkssel).*qq(rowsel,:)) );
            PXX(:,blkssel)=PXX(:,blkssel)*stm';
        end; % for i=varindx
        PsiDot=stm*PsiDot;
        PXX=stm*PXX;
        OldX=XX;
        XX=stm*XX;
        for i=1:nvart,                          % for vars in train
            j=tvar(i); blkssel=j*nx+offcols; trnblk=i*nx+offcols;
            PW(:,blkssel)=PW(:,blkssel)+Psi'*pqq(rowsel,trnblk)*Psi;
            PYY(j)=PYY(j)+sum(sum( OldX.*pqq(rowsel,trnblk) ))+pss(ssel,i);
            PsiDot(:,blkssel)=PsiDot(:,blkssel)+ptt(rowsel,trnblk)*Psi ;
            rrdot=XX*ptt(rowsel,trnblk)';
            PXX(:,blkssel)=PXX(:,blkssel)+rrdot*rrdot'+prr(rowsel,trnblk);
        end; % for i=tvar
        WW=WW+qtemp*Psi;
        Psi=stm*Psi;
        XX=XX*stm'+rr(rowsel,:);
        fprintf('T'); fprintf(int2str(ssel));
        t2go=t2go-dt;
        elaptime=elaptime+dt;
        t2go(i1)=t1;
    else,
        % next event NOT A TRAIN EVENT
        isp=-sptr(i1,spindx(i1));               % event identifier
        if isp>0,                               % isp points to DS block
            %
            DISCRETE EVENT
            blkssel=isp*nx+offcols;
            rowsel=find(any(DS(:,blkssel)~-eye(nx)));
            if length(rowsel)<1, rowsel=1; end;    % guard for rowsel=eye
            stm=DS(rowsel,blkssel);
            PsiDot(rowsel,:)=stm*PsiDot;
            PXX(rowsel,:)=stm*PXX;

```

```

for i=varindx; % for all variables
    inx=(i-1)*nx; blksel=inx+cols; colsel=inx+rowsel;
    PXX(:,colsel)=PXX(:,blksel)*stm';
end; % for i=varindx
XX(rowsel,:)=stm*XX;
varsel=find((vptr(:,1)==isp)');
for i=varsel, inx=(i-1)*nx; % variables in DS block
    blksel=inx+cols; ir=vptr(i,2); ic=vptr(i,3); iro=inx+ir;
    PsiDot(ir,blksel)=PsiDot(ir,blksel)+Psi(ic,:);
    PXX(:,iro)=PXX(:,iro)+XX(:,ic);
    PXX(ir,blksel)=PXX(ir,blksel)+XX(:,ic)';
end; % for i=varsel
Psi(rowsel,:)=stm*Psi;
XX(:,rowsel)=XX*stm';
XX(rowsel,rowsel)=XX(rowsel,rowsel)+x0(rowsel,rowsel);
fprintf('D'); fprintf(int2str(isp));
else, % isp must be < 0 % i.e. continuous segment
    t2go(i1)=t2go(i1)-isp; % increase that t2go element
    fprintf('/');
end; %if isp>0 ... else ...
end; % if (i1==fastseq ... else ..
spindx(i1)=spindx(i1)+1; % increment sequence index
else, % if (t1<= 0) i.e. t1>0 => time passes, continuous segment

```

%

## CONTINUOUS TIME TRANSITIONS

```

ABt1=AB*t1;
At1=ABt1(phirows,phirows);
tempwq=[-ABt1',w0(phicols,phicols)*t1;zeros(ABwide), ABt1];
tempxr=[-At1,x0(phirows,phirows)*t1;zeros(nxc), At1'];
if nargout>6 & length(cvarsel)>0,
    ABdot=zeros(ABwide); i=cvarsel(1); ABdot(vptr(1,2),vptr(1,3))=t1;
    tempwq=[tempwq,[zeros(ABwide);ABdot];zeros(ABwide,2*ABwide),ABt1];
    tempxr=[tempxr,[zeros(nxc);ABdot(phirows,phirows)];...
        zeros(nxc,2*nxc),At1];
end; % if nargout>6 & length(cvarsel)>0
tempwq=expm(tempwq);
tempxr=expm(tempxr);
phix=tempwq(w2cols,w2cols);
phi=phix(phirows,:);
Qt1=w0*t1;
Qt1(phicols,phicols)=phix'*tempwq(phicols,w2cols);
qtemp=Psi'*Qt1;
YY=YY+sum(sum(XX.*Qt1));
Rt1=phi(:,phirows)*tempxr(phirows,x2cols);
inx=0;
for i=varindx;
    blksel=inx+cols;
    inx=inx+nx;

```

```

qddot=qtemp*PsiDot(:,blkssel);
PWW(:,blkssel)=PWW(:,blkssel)+qddot+qddot';
PYY(i)=PYY(i)+sum(sum(PXX(:,blkssel).*Qt1));
PXX(:,inx+offprows)=PXX(:,inx+offpcols)*phi';
end; % for i=varindx
XX(phirows,:)=phi*XX(phicols,:);
PsiDot(phirows,:)=phi*PsiDot(phicols,:);
PXX(phirows,:)=phi*PXX(phicols,:);
for j=1:length(cvarsel), % cont. vars. (not tested)
    jj=cvarsel(j);
    inx=block(jj);
    blkssel=inx+cols;
    blkrow=inx+phirows;
    blkcol=inx+phicols;
    phidot=tempwq(phirows,w3cols);
    PsiDot(phirows,blkssel)=PsiDot(phirows,blkssel)+phidot*Psi(phicols,:);
    rrdot=phi(:,phirows)'*tempxr(phirows,x3cols);
    rrdot=rrdot+rrdot';
    PXX(phirows,blkrow)=PXX(phirows,blkrow)+rrdot;
    rrdot=XX(:,phicols)*phidot';
    PXX(:,blkrow)=PXX(:,blkrow)+rrdot;
    PXX(phirows,blkssel)=PXX(phirows,blkssel)+rrdot';
    qddot=phix'*tempwq(phicols,w3cols);
    qddot=qddot+qddot';
    PWW(phicols,blkcol)=PWW(phicols,blkcol)+qddot;
    if j<length(cvarsel);
        ABdot=zeros(ABwide); i=cvarsel(j+1);
        ABdot(vptr(i,2),vptr(i,3))=t1;
        tempwq(v2cols,v3cols)=ABdot;
        tempxr(x2cols,x3cols)=ABdot(phirows,phirows);
        tempwq=expm(tempwq);
        tempxr=expm(tempxr);
    end; % if j<length(cvarsel)
end; % for i=1:nvar
VW=VW+qtemp*Psi;
Psi(phirows,:)=phi*Psi(phicols,:);
XX(:,phirows)=XX(:,phicols)*phi';
XX(phirows,phirows)=XX(phirows,phirows)+Rt1;
elaptime=elaptime+t1;
t2go=t2go-t1;
fprintf('C. ');
end; % if t1 <=0 ... else ...
[t1,i1]=min(t2go); % next event & its sequence
end; % while elaptime+t1<BTP % no more discretos in BTP

% Clear large temporary arrays
clear qq; clear rr; clear ss; clear tt;
clear pqq; clear prr; clear pss; clear ptt;

```

```

clear phi;      clear phidot;  clear OldX
clear tempwq;   clear tempxr;   clear st;
clear phi;      clear phidot;   clear stm;      clear phix;
clear ABt1;     clear At1;      clear Rt1;      clear Qt1;
clear rrdot;    clear qtemp;    clear qqdot;
clear blksel;   clear rowsel;   clear trnblk;   clear varsel;

if nargout<5; return; end;
%*****End of Psi & PsiDot Calculation*****

%----- Eigen Calculations-----

[P,Lam]=eig(Psi);
Lam=diag(Lam);                                     % convert to vector
for i=1:nx, P(:,i)=P(:,i)/norm(P(:,i)); end;      % normalize eigenvectors
fprintf(' ', Eigs, Partial: ');
if nargout<7, return, end;

%-----Jacobian Calculations-----
PLam=zeros(nx,nvar);                               % allocate space for arrays
PP=zeros(nx,nx*nvar);
temp0=zeros(nx,nx*nvar);
altcol=0:nx:(nx*nvar-1);                          % column offsets for nx by nx*nvar arrays
L=0;  omeg=0;                                       % initialize before first pass

% precompute all PsiDot*P(:,i) vectors
colsel=cols;
for i=varindx, temp0(:,colsel)=PsiDot(:,colsel)*P; colsel=colsel+nx; end;

for i=cols,
    if (i>1)&(omeg~=0)&(conj(Lam(i))==L),           % index to eigenvalue/vector
        % short cut for conjugates
        PLam(i,:)=conj(PLam(i-1,:));
        PP(:,altcol+i)=conj(PP(:,altcol+i-1));
    else, % if (i>1)&(conj(Lam(i))=L)              % normal path
        colsel=i+altcol;
        p=P(:,i);
        L=Lam(i);
        omeg=imag(L);

        % precompute temp1=pinv(L*eye+(eye-p*p')*Psi)
        temp1=L*eye(nx)-Psi+p*(p'*Psi);
        [U,S,V]=svd(temp1);
        S=diag(S);
        tol=nx*S(1)*eps;
        ir=sum(S>tol);
        rsel=1:ir;
        S=diag(ones(ir,1)./S(rsel));
        temp1=V(:,rsel)*S*U(:,rsel)';
    end
end

```

```
% precompute temp1=pinv(L*eye+(eye-p*p')*Psi)*(I-p*p')
temp1=temp1-(temp1*p)*p';

% compute p_dot_ortho and store in PP
PP(:,colsel)=temp1*temp0(:,colsel);

% compute lamdot = p'*PsiDot*p + p'*Psi*p_dot_ortho
PLam(i,:)=p'*(temp0(:,colsel)+Psi*PP(:,colsel));

if omeg~=0, % add correction to p_dot_ortho for complex EV's
    sig=real(L);
    temp2=Psi*imag(p)-sig*imag(p)+omeg*real(p);      % RHS of eqn 9
    [maxcompr,maxindx]=max(abs(temp2));
    maxcompr=temp2(maxindx);                          % biggest component
    maxcompl=real(temp0(maxindx,colsel));
    maxcompl=maxcompl - real(PLam(i,:))*real(p(maxindx)); % same
    maxcompl=maxcompl + imag(PLam(i,:))*imag(p(maxindx)); % component
    maxcompl=maxcompl+Psi(maxindx,:)*real(PP(:,colsel)); % for nvar
    maxcompl=maxcompl - sig*real(PP(maxindx,colsel));     % LHS eqns
    maxcompl=maxcompl -omeg*imag(PP(maxindx,colsel));
    gamma=sqrt(-1)*(maxcompl/maxcompr);                 % coefficient
    PP(:,colsel)=PP(:,colsel)+p*gamma;                   % true P_Dot vectors
end; % if omeg~=0 ... else ...
end; % if (i>1)&(conj(Lam(i))==Lam(i-1))
fprintf(int2str(i));                                     % something to watch
end; % for i=i:nx

fprintf(' DONE. ');
fprintf(' SR= %g',max(abs(Lam)) );
fprintf('\n');
return;
% end of function psi_plus $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

%*****
%
% OUTPUT: Psi = BTP state transition matrix
%          XX  = E(xx') at BTP start
%          WW  = BTP weighting matrix at BTP start
%          YY  = cost correction for noise in current BTP
%          Lam = column vector of Psi's eigenvalues
%          P   = matrix of Psi's unit eigenvectors
%          PXX = partials XX (block row) wrt theta
%          PWX = partials WW (block row) wrt theta
%          PYY = partials YY wrt theta
%          PLam = Jacobian martix (d Lam / d theta) (nx by nvar)
%          PP, row of (d P / d theta) block matrices (nx by nvar*nx)
% INPUT: t2go = times from BTP start to start of each sequence (>0).
```

```

%      AB,DS,vptr,and sptr: (see 'build.m' and 'update.m')
%      con: (see 'check.m')
%      x0 = process noise covariance
%      w0 = static weighting matrix
%      qq      =      [ W1 ; W2 ]
%      rr      =      [ X1 ; X2 ]
%      ss      =      [ Y1 ; Y2 ]
%      tt      =      [ Psi1 ; Psi2 ]
%      pqq      =      [ W1_dot_1 ... W1_dot_nvart ;
%                      W2_dot_1 ... W2_dot_nvart ]
%      prr      =      [ X1_dot_1 ... X1_dot_nvart ;
%                      X2_dot_1 ... X2_dot_nvart ]
%      pss      =      [ Y1_dot_1 ... Y1_dot_nvart ;
%                      Y1_dot_2 ... Y2_dot_nvart ]
%      ptt      =      [ Psi1_dot_1 ... Psi1_dot_nvart ;
%                      Psi2_dot_1 ... Psi2_dot_nvart ]
%
%      Where Psi1, W1, X1 and Y1 apply to the shorter train (k-1 periods),
%      and Psi2, W2, X2, and Y2 apply to the longer train (k periods).
%      The '_dot_i' notation indicates the partial with respect to the i'th
%      variable (theta element) in the fast sequence (nvart such variables).
%
%***** STATE TRANSITION MATRIX PARAMETERS*****
%
%      These parameters are defined for any time period:
%      Psi = State Transition Matrix for the period
%      PsiDot = partial of Psi wrt a variable parameter (theta element)
%      W = weighting:  $x(0)'W x(0) = \int_{\text{period}} x(t)'W_0 x(t) dt$ 
%      W_dot = partial of W wrt a variable parameter (theta element)
%      X =  $E\{x(f)x(f)'\}$  from process noise (x0) during period (if x(0)=0)
%      X_dot = partial of X wrt a variable parameter (theta element)
%      Y =  $E\{\int_{\text{period}} x(t)'W_0 x(t) dt\}$  if x(0)=0. (scalar)
%      Y_dot = partial Y wrt a variable parameter (theta element)
%
%      RECURSIVE ALGORITHM FOR COMPUTING STM PARAMETERS
%      Suppose:  $\Psi(t_f,t_0) = T(n)*T(n-1)* \dots *T(2)*T(1)$ , where T(i) is
%      the state transition matrix (stm) for the i'th segment of the period.
%      Let Q(i),R(i), and S(i) represent W,X, and Y, respectively, for T(i).
%
%      Initialize:
%      Psi(0)=I,      PsiDot(0)=[0,0,...,0,0]      These may be considered
%      W(0)=0;      W_dot(0)=[0,0,...,0,0]      correct values for T1=I,
%      X(0)=0;      X_dot(0)=[0,0,...,0,0]      a null transition with
%      Y(0)=0;      Y_dot(0)=[0,0,...,0,0]      zero elapsed time.
%
%      Recursions:
%      Psi(i)=T(i)*Psi(i-1),
%      PsiDot(i)=T(i)*PsiDot(i-1)+T_dot_(i)*Psi(i-1),

```

```
%
W(i)=W(i-1) + Psi(i-1)*Q(i)*Psi(i-1)
W_dot(i)=W_dot(i-1) + PsiDot(i-1)*Q(i)*Psi(i-1)...
          +Psi(i-1)'*Q_dot(i)*Psi(i-1) + Psi(i-1)'*Qi*PsiDot(i-1)
X(i)=T(i)*X(i-1)*T(i)' + R(i)
X_dot(i)=T_dot(i)*X(i-1)*T(i)' + T(i)*X_dot(i-1)*T(i)'...
          +T(i)*X(i-1)*T_dot(i)' + R_dot(i)
Y(i)=Y(i-1) + X(i-1).*Q(i)
Y_dot(i)=Y_dot(i-1) + X_dot(i-1).*Q(i) + X(i-1).*Q_dot(i)

%
% Then, Psi(n), PsiDot(n), W(n), W_dot(n), X(n), X_dot(n),
% Y(n) and Y_dot(n) apply to the period covered by Psi.
%
% *****
%
% Train concept: The fastest sampler has frequent bursts or trains of
% k or k-1 consecutive short sample periods where:
%     k=trunc(2'nd shortest sample period/shortest sample period).
% Data for these fixed STM's are precalculated and stored in 'trndat'
% to avoid subsequent reduce redundant calculations. 'trndat' includes
% the discrete events on both ends of the fast 'sample train.'
%
% The precalculated trndat and cc parameters are defined as:
%
% qq      = [ W1 ; W2 ]
% rr      = [ X1 ; X2 ]
% ss      = [ Y1 ; Y2 ]
% tt      = [ Psi1 ; Psi2 ]
%
% pqq     = [ W1_dot_1 ... W1_dot_nvart ;
%            W2_dot_1 ... W2_dot_nvart ]
% prr     = [ X1_dot_1 ... X1_dot_nvart ;
%            X2_dot_1 ... X2_dot_nvart ]
% pss     = [ Y1_dot_1 ... Y1_dot_nvart ;
%            Y1_dot_2 ... Y2_dot_nvart ]
% ptt     = [ Psi1_dot_1 ... Psi1_dot_nvart ;
%            Psi2_dot_1 ... Psi2_dot_nvart ]
%
% Where Psi1, W1, X1 and Y1 apply to the shorter train (k-1 periods),
% and Psi2, W2, X2, and Y2 apply to the longer train (k periods).
% The '_dot_i' notation indicates the partial with respect to the i'th
% variable (theta element) in the fast sequence (nvart such variables).
%
% end of function Psi_plus $$$$$$
```

### E.3.8 Precomputation for psi2plus: trains2.m

```
function [QQ,RR,SS,TT,QQdot,RRdot,SSdot,TTdot]=...
```

```

trains2(w0,x0,AB,DS,seqptr,varptr,consts,block,tvarsel,cvartsel);

% *****
% ** This version is limited to nseq=1 or 2 only. **
% *****
% See acstgr42.m for additional comments and discussion

fprintf(' Train2: step: ');

nx=consts(1);      nxc=consts(2);      nxs=consts(3);
nvar=consts(5);    nvarc=consts(6);    nseq=consts(7);
seqlen=consts(8);  keyseq=consts(9);    fastseq=consts(11);
ABwide=nxc+nx;     cycles=round(consts(12)/consts(13)); % trainlen/period
nvar=length(tvarsel);      nxv=nx*nvar;
nvarct=sum(varptr(tvarsel,1)==0);

%          indices for array operations
cols=1:nx;         ccols2=cols+nx;         offcols=cols-nx;
phirows=1:nxc;     phi2rows=phirows+nxc;    phi3rows=phi2rows+nxc;
phicols=1:ABwide;  phi2cols=phicols+ABwide;  phi3cols=phi2cols+ABwide;
varindx=1:nvar;

%          COMPUTE PARAMETERS FOR ONE CYCLE OF FASTSEQ.

%          Put first event in train data
isp=-seqptr(fastseq,1);          % initial event number
blk1=isp*nx+offcols;
rowssel=find(any(DS(:,blk1)~=eye(nx))); % non-identity rows
Q=zeros(nx);                    Qdot=zeros(nx,nxv); % initial W/Wdot
R=zeros(nx);                    Rdot=zeros(nx,nxv); % initial X/Xdot
S=0;                            Sdot=zeros(1,nvar); % initial Y/Ydot
T=DS(:,blk1);                   Tdot=zeros(nx,nxv); % initial Psi/PsiDot
varsel1=find(varptr(:,1)~=isp); % variables in fastseq
for i=varsel1;                  % step thru fastseq vars
    ir=varptr(i,2); ic=varptr(i,3)+block(1); % Tdot row/column
    Tdot(ir,ic)=1;              % Tdot(ir,ic)=1
end; % for i=varsel1
fprintf(int2str(1));            % progress display

%          Include rest of first cycle in train data
for j=2:seqlen;
    isp=-seqptr(fastseq,j);      % event identifier
    if isp>0,                    % isp points to DS block
        %          DISCRETE EVENT
        blk=isp*nx+offcols;      % index to DS
        rowssel=find(any(DS(:,blk)~=eye(nx))); % non-identity rows
    end
end

```



```

stm=DS(rowssel,blk); % new stm
Tdot(rowssel,:)=stm*Tdot; % PsiDot=stm*PsiDot
R(rowssel,:)=stm*R; % X=stm*X
Rdot(rowssel,:)=stm*Rdot; % Xdot=stm*Xdot
for i=varindx, % for all Xdot blocks
    inx=(i-1)*nx;
    Rdot(:,inx+rowssel)=Rdot(:,inx+cols)*stm'; % Xdot=[stm*Xdot]*stm'
end; % for i=varindx
varsel=find(isp==varptr(:,1)'); % find vars in stm
for i=varsel, % for variables in stm, add Tdot terms
    blk=block(i)+cols;
    ir=varptr(i,2); ic=varptr(i,3);
    Tdot(ir,blk)=Tdot(ir,blk)+T(ic,:); % PsiDot=stm*PsiDot+stmDot*Psi
    ico=block(i)+ir; % Xdot=stm*Xdot*stm'+stm*X*stmDot'+stmDot*X*stm'
    Rdot(:,ico)=Rdot(:,ico)+R(:,ic);
    Rdot(ic,blk)=Rdot(ic,blk)+R(:,ic)';
end; % for i=varsel
T(rowssel,:)=stm*T; % Psi=stm*Psi
R(:,rowssel)=R*stm'; % X=[stm*X]*stm' + process noise
R(rowssel,rowssel)=R(rowssel,rowssel)+x0(rowssel,rowssel);
else, % isp must be < 0 % continuous time only
% CONTINUOUS TIME TRANSITION
t1=-isp; ABt1=AB*t1; At1=ABt1(phirows,phirows);
tempwq=[-ABt1',w0(phicols,phicols)*t1;zeros(ABwide),ABt1];
tempxr=[-At1,x0(phirows,phirows)*t1;zeros(nxc),At1'];
if nvarc>0,
    ABdot=zeros(ABwide); i=cvarsel(1);
    ABdot(varptr(i,2),varptr(i,3))=t1;
    tempwq=[tempwq,[zeros(ABwide);ABdot];zeros(ABwide,2*ABwide),ABt1];
    tempxr=[tempxr,[zeros(nxc);ABdot(phirows,phirows)];...
        zeros(nxc,2*nxc),At1];
end; % if nvarc>0
tempwq=expm(tempwq);
tempxr=expm(tempxr);
phix=tempwq(phi2cols,phi2cols); % phi extended
phi=phix(phirows,:); % phi=[Phi(t1),Gamma(t1)];
Qt1=w0*t1; % Qt1=\int_0^{t1} B'exp(A't) w0 exp(At) B dt
Qt1(phicols,phicols)=phix'*tempwq(phicols,phi2cols);
% Rt1=\int_0^{t1} exp(At) x0 exp(A't) dt
Rt1=phi(:,phirows)*tempxr(phirows,phi2rows);
S=S+sum(sum(R.*Qt1)); % delta cost correct
qtemp=T'*Qt1; % Psi'*Q
qqdot=qtemp*Tdot; % Psi'*Q*PsiDot
Qdot=Qdot+qqdot; % Wdot=Wdot+Psi'*Q*PsiDot
for i=varindx; % column operations
    inx=(i-1)*nx; blk=inx+cols; colsel=inx+phicols; rowssel=inx+phirows;
    % Wdot=[Wdot+Psi'*Q*PsiDot]+PsiDot'*Q*Psi
    Qdot(:,blk)=Qdot(:,blk)+qqdot(:,blk)';

```

```

    Sdot(i)=Sdot(i)+sum(sum(Rdot(:,blk).*Qc1));
    Rdot(:,rowssel)=Rdot(:,colsel)*phi';           % Xdot=Xdot*stm'
end; % for i=varindx
OldR=R;
R(phirows,:)=phi*R(phicols,:);                   % X=stm*X
Rdot(phirows,:)=phi*Rdot(phicols,:);              % Xdot=stm*[Xdot*stm']
Tdot(phirows,:)=phi*Tdot(phicols,:);
    % INCLUDE: T_dot, Q_dot, and R_dot TERMS (IF ANY)
for j=1:nvarct,                                     % cont. vars. (not tested)
    i=min(find(cvarsel(j)==tvarsel));
    inx=block(cvarsel(j));
    blk=inx+cols; rowssel=inx+phirows; colsel=inx+phicols;
    phidot=tempwq(phirows,phi3cols);                % T_dot
    Tdot(phirows,blk)=Tdot(phirows,blk)+phidot*T(phicols,:);
    rtemp=phi(:,phirows)*tempxr(phirows,phi3rows);   % R_dot
    Rdot(phirows,rowssel)=Rdot(phirows,rowssel)+rtemp; % Xdot+Rdot
    rrdot=R(:,phicols)*phidot';                      % stm*X*stmDot'
    Rdot(:,rowssel)=Rdot(:,rowssel)+rrdot;
    Rdot(phirows,blk)=Rdot(phirows,blk)+rrdot';
    qqtemp=phix'*tempwq(phicols,phi3cols);           % Q_dot
    Sdot(i)=Sdot(i)+sum(sum(OldR(phicols,phicols).*qqtemp));
    tqdot=T(:,phicols)*qqtemp;
    Qdot(:,blk)=Qdot(:,blk)+tqdot*T(phicols,:);
    if j<nvarc;                                       % set up next cvar, if any
        ABdot=zeros(ABwide); i=cvarsel(j+1);
        ABdot(varptr(i,2),varptr(i,3))=t1;
        tempwq(phi2cols,phi3cols)=ABdot;
        tempxr(phi2rows,phi3rows)=ABdot(phirows,phirows);
        tempwq=expm(tempwq);
        tempxr=expm(tempxr);
    end; % if i~=cvarsel(())
end; % for i=1:nvarc
Q=Q+qtemp*T;                                       % W=W+stm'*Q*stm
T(phirows,:)=phi*T(phicols,:);                   % Psi=stm*Psi
R(:,phirows)=R(:,phicols)*phi';                  % X=[stm*X]*stm'
R(phirows,phirows)=R(phirows,phirows)+Rti;       % X=[stm*X*stm'] + R
end; % if isp>0 ... else ...
fprintf(int2str(j));
end; % for j=2:seqlen                               % no more discreties in period

% EXTEND TO 'CYCLES' & 'CYCLES+1' PERIODS AND ADD FINAL DISCRETE EVENT
fprintf(' ', cycle: ');
Psi=T;      PsiDot=Tdot;
X=R;      Xdot=Rdot;
W=Q;      Wdot=Qdot;
cc=S;      ccdot=Sdot;
stm=DS(rowssel1,blk1);                             % stm for final event
fprintf(int2str(1));

```

```

loopcount=[cycles,2]; % set number of loop iterations
for case=1:2; % 1: first cycles periods; 2: last period
    offcyc=(case-1)*(cycles-1);
    for i=2:loopcount(case),
        cc=cc+sum(sum(X.*Q))+S;
        qtemp=Psi'*Q;
        qqdot=qtemp*PsiDot;
        qqdot=Psi'*Qdot;
        Wdot=Wdot+qqdot;
        OldX=X;
        X=T*X;
        OldXdot=Xdot;
        Xdot=T*Xdot;
        PsiDot=T*PsiDot;
        for j=varindx,
            jnx=(j-1)*nx; blk=jnx+cols;
            Wdot(:,blk)=Wdot(:,blk)+qqdot(:,blk)+qqdot(:,blk)*Psi;
            ccdot(j)=ccdot(j)+sum(sum(OldXdot(:,blk).*Q...
                +OldX.*Qdot(:,blk) ))+Sdot(j);
            rtemp=X*Tdot(:,blk)';
            Xdot(:,blk)=Xdot(:,blk)*T' + rtemp + rtemp'*Rdot(:,blk);
            PsiDot(:,blk)=PsiDot(:,blk)+Tdot(:,blk)*Psi;
        end; % for j=varindx
        W=W+qtemp*Psi;
        Psi=T*Psi;
        X=X*T'+R;
        fprintf(int2str(i+offcyc));
    end;
    base=(case-1)*nx; colsel=base+cols; rowsel=base+rowsel;
    QQ(colsel,:)=W; QQdot(colsel,:)=Wdot;
    SS(case)=cc; SSdot(case,:)=ccdot;
    RR(colsel,:)=X; RR(rowsel,:)=stm*X;
    RRdot(colsel,:)=Xdot; RRdot(rowsel,:)=stm*Xdot;
    TTdot(colsel,:)=PsiDot; TTdot(rowsel,:)=stm*PsiDot;
    for i=varindx, inx=(i-1)*nx;
        RRdot(colsel,inx+rowsel)=RRdot(colsel,inx+cols)*stm';
    end; % for i=varindx
    for i=varsel, blk=block(i)+cols;
        ir=varptr(i,2); ic=varptr(i,3); iro=base+ir; ico=block(i)+ir;
        TTdot(iro,blk)=TTdot(iro,blk)+Psi(ic,:);
        RRdot(colsel,ico)=RRdot(colsel,ico)+RR(colsel,ic);
        RRdot(iro,blk)=RRdot(iro,blk)+RR(colsel,ic)';
    end; % for i=varsel
    TT(colsel,:)=Psi; TT(rowsel,:)=stm*Psi;
    RR(colsel,rowsel)=RR(colsel,:)*stm';
    RR(rowsel,rowsel)=RR(rowsel,rowsel)+x0(rowsel1,rowsel1);
    fprintf('/');

```

```
end; % for case=1:2
fprintf('DONE.\n');
```

```
% end of function trains2.m $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

### E.3.9 STM Only: psi2.m

```
function [Psi,XX,WW,YY]=...
    psi2(t2go,AB,DS,sptr,con,x0,w0,qq,rr,ss,tt);

% Compute STM and related matrices.
% See function psi2plus for detailed explanation.

%                UNPACK CONSTANTS
nxc=con(1);          nxc=con(2);          nxs=con(3);
nvar=con(5);          nvarc=con(6);          nseq=con(7);
seqlen=con(8);        BTP=con(10);          fastseq=con(11);
trainlen=con(12);     speriod=con(13);      ABwide=nxc+nxs;
%                CREATE INDICIES FOR ARRAY OPERATIONS
cols=1:nx;           offcols=cols-nx;       varindx=1:nvar;
phirows=1:nxc;        x2cols=phirows+nxc;
phicols=1:ABwide;     w2cols=phicols+ABwide;
offpcols=phicols-nx;  offprows=phirows-nx;
%                VERIFY t2go IN RANGE
if length(t2go)~=nseq, errmsg('Extra/missing phase times.'): end;
if (any(t2go<0)), errmsg('Negative time to go in psi plus.'): end;

%-----Set Pointers for Initial Discrete Events-----
%    t2go(i) will be the time from BTP start to the first discrete
%    event for each sequence. 'sptr(i,spindx(i))' defines the event.
done=1;  spindx=ones(t2go)*seqlen;  laststep=zeros(t2go);
while done>0;
    % completion flag
    for i=1:nseq,laststep(i)=sptr(i,spindx(i));end; % index to last event
    laststep=laststep.*(laststep>0); % mask discrete events
    backstep=(laststep<t2go); % boolean
    done=sum(backstep); % 0 if done
    t2go=t2go-backstep.*laststep; % adjust t2go
    t2go=t2go.*(t2go>0); % eliminate any negative times
    spindx=spindx-backstep; % adjust index
    spindx=spindx+seqlen*(spindx<1); % fix possible wrap-around
end; % while done>0
spindx=spindx+1; % set to current event
spindx=spindx-(spindx>seqlen)*seqlen; % fix wrap-around

%-----Compute Psi, XX, WW, YY-----
% Psi = psi, the state transition matrix for the specified t2go.
% XX = E xx' at BTP end from process noise during BTP
```

```

% WW = matrix s.t.  $x_0' * WW * x_0$  is the cost for  $x_0$  error at BTP start
% YY = cost from process noise during BPT
% All used as accumulators for intermediate results during recursions.

%              INITIALIZE FOR FIRST EVENT
done=1;                elaptime=0;                % time into BTP
zerotol=BTP*eps;
[t1,i1]=min(t2go);      % time to first event
isp=-sptr(i1,spindx(i1)); % identify first stm
XX=zeros(nx); WW=zeros(nx); YY=0; Psi=eye(nx); % initialize outputs
%              QUICKIE INITIAL DISCRETE EVENT IF PRACTICAL
if isp>0,                % first event discrete
    blksel=isp*nx+offcols;
    Psi=DS(:,blksel);
    spindx(i1)=spindx(i1)+1;
end; % if isp>0

%              MAIN STATE TRANSITION MATRIX (stm) CALCULATION LOOP
while done>0,            % main Psi loop
    if (elaptime+t1+zerotol)>=BTP, t1=BTP-elaptime;done=0;end; %last step
    spindx=spindx-(spindx>seqlen)*seqlen; % fix any wrap-around
    if t1<=0,             % t1=0 =>next event now
        st=sort(t2go);    % get next event time too
        t2=st(2);         % time to second event
        if (i1==fastseq)&(t2>=trainlen)&(spindx(i1)==1)&...
            ((trainlen+elaptime)<BTP), % next event is a train
            %              TRAIN EVENT
            rowssel=cols; ssel=1; dt=trainlen; % use short train indices
            if (t2>=(dt+aperiod))&((dt+elaptime+aperiod)<BTP), % k cycles
                rowssel=rowssel+nx; ssel=2; dt=dt+aperiod;% use long train indices
            end;
            stm=tt(rowssel,:);
            YY=YY+sum(sum(XX.*qq(rowssel,:)))+ss(ssel);
            qtemp=Psi'*qq(rowssel,:);
            XX=stm*XX;
            WW=WW+qtemp*Psi;
            Psi=stm*Psi;
            XX=XX*stm'+rr(rowssel,:);
            t2go=t2go-dt;
            elaptime=elaptime+dt;
            t2go(i1)=t1;
        else,              % next event NOT A TRAIN EVENT
            isp=-sptr(i1,spindx(i1)); % event identifier
            if isp>0,        % isp points to DS block
                %              DISCRETE EVENT
                blksel=isp*nx+offcols;
                rowssel=find(any(DS(:,blksel)~=eye(nx)));
                if length(rowssel)<0, rowssel=1; end; % guard for rowssel=eye
            end;
        end;
    end;
end;

```

```

        stm=DS(rowssel,blkssel);
        XX(rowssel,:)=stm*XX;
        Psi(rowssel,:)=stm*Psi;
        XX(:,rowssel)=XX*stm';
        XX(rowssel,rowssel)=XX(rowssel,rowssel)+x0(rowssel,rowssel);
    else, % isp must be < 0          % i.e. continuous segment
        t2go(i1)=t2go(i1)-isp;      % increase that t2go element
    end; %if isp>0 ... else ...
end; % if (i1==fastseq ... else ..
spindx(i1)=spindx(i1)+1;          % increment sequence index
else, % if (t1<= 0)   i.e. t1>0 => time passes, continuous segment

%
CONTINUOUS TIME TRANSITIONS
ABt1=AB*t1;
At1=ABt1(phirows,phirows);
tempwq=[-ABt1',w0(phicols,phicols)*t1;zeros(ABwide), ABt1];
tempxr=[-At1,x0(phirows,phirows)*t1;zeros(nxc), At1'];
tempwq=expm(tempwq);
tempxr=expm(tempxr);
phix=tempwq(w2cols,w2cols);
phi=phix(phirows,:);
Qt1=w0*t1;
Qt1(phicols,phicols)=phix'*tempwq(phicols,w2cols);
qtemp=Psi'*Qt1;
YY=YY+sum(sum(XX.*Qt1));
Rt1=phi(:,phirows)*tempxr(phirows,x2cols);
XX(phirows,:)=phi*XX(phicols,:);
WW=WW+qtemp*Psi;
Psi(phirows,:)=phi*Psi(phicols,:);
XX(:,phirows)=XX(:,phicols)*phi';
XX(phirows,phirows)=XX(phirows,phirows)+Rt1;
elaptime=elaptime+t1;
t2go=t2go-t1;
end; % if t1 <=0 ... else ...
[t1,i1]=min(t2go);          % next event & its sequence
end; % while elaptime+t1<BTP % no more discretes in BTP

% end function psi2 *****

```

### E.3.10 Linear Search: linsrch2.m

```

function optstp=...
linsrch2(dir,istep,icost,btime,w0,x0,AB,DS,itheta,seqptr,varptr,consts);

% linsrch finds the step (in the specified direction) to minimize cost.
% icost = cost with alpha=0, and itheta+optstp*dir ==>theta for min cost

```

```

nx=consts(1);
GoodEnuf=.10; % 10 pct convergence tolerance
tol=1e-8*icost+eps*nx*nx*nx; % minimum cost improvement
zerostep=1e-10; % threshold for effectively zero step
count=0; % iteration counter
maxcount=30; % Loop quits after this many iterations
gam2=2/(1+sqrt(5)); % golden section ratio (.62 approx)
gam1=1-gam2; % golden section ratio (.38 approx)
scopestep=5; % factor to shrink/stretch search region
istep=real(istep); % make sure it's real
optstp=istep; % load some initial value

fprintf(' linsrch2:');

% Part 1: find region containing minimum (assuming cost unimodal).

alpha=zeros(4,1); % vector of step sizes
cost=zeros(4,1); % vector of corresponding costs
cost(1)=icost;
alpha(2)=istep;
ttheta=itheta+alpha(2)*dir; % update temporary theta
[AB,DS]=update(AB,DS,ttheta,varptr,consts); % update AB & DS
cost(2)=acst2(btime,w0,x0,AB,DS,seqptr,consts);
alpha(3)=alpha(2);
cost(3)=cost(2);
fprintf(' linsrch2: initial end point = %g',alpha(2));
fprintf(', cost= %3.5g \n',cost(2));
fprintf('E');

if cost(2)<cost(1), % cost still decreasing
    case=1; factor=scopestep; ctrpt=2; refend=3; symb='>';
else, % if cost(4)<cost(1) % istep is past minimum
    case=2; factor=1/scopestep; ctrpt=3; refend=1; symb='<';
end;

while cost(ctrpt)>=cost(refend); % step until center is lower
    count=count+1;
    if count>maxcount, optstp=-eps; break; end; % failure
    alpha(2)=alpha(3);
    cost(2)=cost(3);
    alpha(3)=alpha(2)*factor; % new alpha(3)
    ttheta=itheta+alpha(3)*dir;
    [AB,DS]=update(AB,DS,ttheta,varptr,consts);
    cost(3)=acst2(btime,w0,x0,AB,DS,seqptr,consts);
    fprintf(symb);
    fprintf(' scoping: alpha = %g',alpha(3));
    fprintf(', cost= %3.5g \n',cost(3));
end; % while cost(ctrpt)>=cost(refend)

```

```

if optstp<=eps; return; end;                                % error return #1
best3=1:3;                                                  % three best points
temp4=ones(4,1)*cost(ctrpt);                                % best cost so far

% Part 2: guarded parabolic/golden section search for minimum

converrr=1+max(cost);
while converrr>=(GoodEnuf*(icost-temp4(1))+tol), % converge to 10 pct
    count=count+1;
    if count>maxcount, optstp=-eps; break; end; % failure
    x=alpha(best3);      y=cost(best3);
    [x,xindx]=sort(x);    y=y(xindx); % put alpha-ascending order
    xspan=x(3)-x(1);      xc=(x(2)-x(1))/xspan;
    X=[1, 1; xc*xc, xc];
    Y=[(y(3)-y(1)); (y(2)-y(1))];
    C=X\Y; % approx: y=C(1)*x^2+C(2)*x+y(1)
    xmin=-C(2)/(2*C(1)); % x estimate for min cost
    ymin=(C(1)*xmin+C(2))*xmin+y(1); % estimate for min cost
    if (2*x(2))>(x(1)+x(3)), xg=gam1; else, xg=gam2; end;
    if abs(xmin-xc)>abs(xg-xc),
        newalpha=x(1)+xg*xspan; case=1; % new alpha if golden section
    else,
        newalpha=x(1)+xmin*xspan; case=2; % new alpha if parabolic fit
    end;
    alpha=[x;newalpha];
    ttheta=itheta+dir*newalpha;
    [AB,DS]=update(AB,DS,ttheta,varptr,consts);
    newcost=acst2(btime,w0,x0,AB,DS,seqptr,consts);
    cost=[y;newcost];
    [temp4,best4]=sort(cost);
    best3=best4(1:3);
    converrr=temp4(4)-temp4(1);
    if case==2, ce=abs(ymin-newcost); converrr=min(converrr,ce);
        %fprintf('          interp: step= ');
        fprintf('P');
    else,
        %fprintf('          golden: step= ');
        fprintf('G');
    end;
    %fprintf('%8.6f, cost= %12.5e, err= %12.5e \n',alpha(4),cost(4),converrr);

end; % while convergence error > 5 percent

optstp=real(alpha(best4(1)));
%fprintf('    Final cost= %3.5g, final step= %3.5g \n',temp4(1),optstp);
fprintf(' Cost= %7.5g, step= %7.5g.\n',temp4(1),optstp);

return;

```



```
% end of function linsrch $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```



# Bibliography

- [Ami80] Naftali Amit. *Optimal Control of Multirate Digital Control Systems*. PhD thesis, Stanford University, Department of Aeronautics and Astronautics, Stanford, CA 94305, July 1980. SUDAAR #523.
- [AP81] Naftali Amit and J. David Powell. Optimal control of multirate systems. In *AIAA, Guidance and Control Conference*, Albuquerque, NM, Aug 1981. Paper No. 81-1797.
- [Ber86] Martin Conrad Berg. *The Design of Multirate Digital Control Systems*. PhD thesis, Stanford University, Department of Aeronautics and Astronautics, Stanford, CA 94305, March 1986. SUDAAR #553.
- [BF75] W. H. Boykin and R. D. Frazier. Analysis of multiloop multirate sampled data systems. *AIAA Journal*, 13(4):453-456, April 1975.
- [BG80] John R. Broussard and Douglas P. Glasson. Optimal multirate flight control design. In *1980 Joint Automatic Control Conference*, AIAA, San Francisco, CA, Aug 1980.
- [BH75] Arthur E. Bryson, Jr. and Yu-chi Ho. *Applied Optimal Control*. Hemisphere Publishing Corp., Washington, 1975.
- [BH84] John R. Broussard and Nesim Halyo. Optimal multi-rate output feedback. In *Proceedings of 23rd Conference on Decision and Control*, pages 926-929, IEEE, Dec 1984.
- [CW66] T. C. Coffey and I. J. Williams. Stability analysis of multiloop, multirate sampled data systems. *AIAA Journal*, 4(12):2178-2190, Dec 1966.

- [DB74] Germund Dahlquist and Åke Björk. *Numerical Methods*. Prentice-Hall, Englewood Cliffs, NJ, 1974.
- [FP80] Gene F. Franklin and J. David Powell. *Digital Control*. Addison-Wesley Publishing Company, Reading, MA, 1980.
- [Gan59] R. F. Gantmacher. *The Theory of Matrices*. Chelsea Publishing Company, New York, N.Y., 1959.
- [GB79] Douglas P. Glasson and John R. Broussard. *Design of Optimal Multirate Estimators and Controllers- Preliminary Results*. Technical Report TIM-1356-1, The Analytic Sciences Corporation, Reading, MA, Aug 1979.
- [Gla83] Douglas P. Glasson. Development and application of multirate digital control. *Control System Magazine*, :2-8, Nov 1983.
- [GMW81] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization*. Academic Press, 1981.
- [GVL83] Gene H. Golub and Charles E. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 1983.
- [Jur67] E. I. Jury. A general z-transform for sampled data systems. *IEEE Transactions on Automatic Control*, AC-12:606-608, Oct 1967.
- [Jur68] E. I. Jury. *Sampled Data Control Systems*. John Wiley & Sons, Inc., New York, 1968.
- [Kai80] Thomas Kailath. *Linear Systems*. Prentice-Hall, Inc., Englewood Cliffs, N. J., 1980.
- [Kai81] Thomas Kailath. *Lectures on Wiener and Kalman Filtering*. Springer-Verlag, Wein - New York, 1981.
- [KB59] R. E. Kalman and J. E. Bertram. A unified approach to the theory of sampling systems. *Journal of the Franklin Institute*, 267:405-436, May 1959.

- [Kok84] Petar V. Kokotovic. Applications of singular perturbation techniques to control problems. *SIAM Review*, 26(4):501-550, Oct 1984.
- [Kra57] G. M. Kranc. Input-output analysis of multirate feedback systems. *IRE Transactions on Automatic Control*, AC-3:21-28, Nov 1957.
- [Len86] Brengt Lennartson. *On the Design of Stochastic Control Systems with Multirate Sampling*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 1986. Technical Report 161.
- [MLBK85] Cleve Moler, John Little, Steve Bangert, and Steve Kleiman. *PC-MATLAB User's Guide*. The MathWorks, Inc., Sherborn, MA, Nov 1985. Version 2.0.
- [MVL78] Cleve Moler and Charles Van Loan. Nineteen dubious ways to compute the exponential of a matrix. *SIAM Review*, 20(4):801-836, Oct 1978.
- [RF58] J. R. Ragazzini and Gene F. Franklin. *Sampled Data Control Systems*. McGraw-Hill, New York, 1958.
- [SBD\*76] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines - EISPACK Guide. Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 2 edition, 1976.
- [SR55] J. Sklansky and J. R. Ragazzini. Analysis of errors in sampled data feedback systems. *AIEE Transactions*, 74(II):65-71, May 1955.
- [Str76] Gilbert Strang. *Linear Algebra and its Applications*. Academic Press, New York, 1976.
- [VL78] Charles Van Loan. Computing integrals involving the matrix exponential. *IEEE Transactions on Automatic Control*, AC-23(3):395-404, Jun 1978.

- [Wal81] Vincent M. Walton. State space stability analysis of multirate-multiloop sampled data systems. In *AAS/AIAA Astrodynamics Specialist Conference*, Aug 1981. Paper 81-201.
- [WH78] R. F. Whitbeck and L. G. Hofmann. *Analysis of Digital Flight Control Systems with Flying Qualities Applications*. Technical Report AFFDL-TR-78-115, Air Force Flight Dynamics Laboratory, Sep 1978. Volume II.